

MIT LIBRARIES

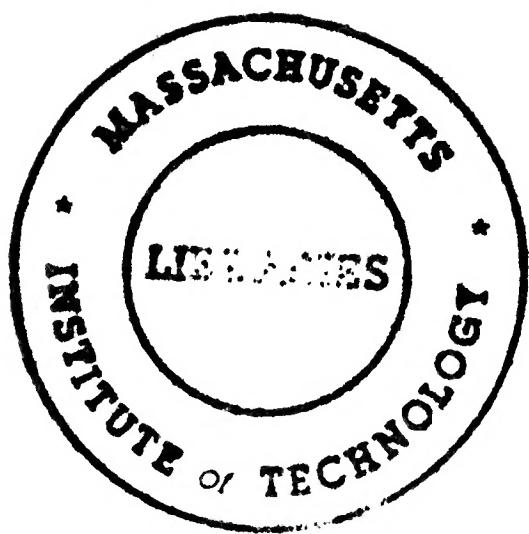


3 9080 02226 2742

*Computer Systems
Series*

***Micropogrammable
Parallel Computer
MUNAP and Its Applications***

Takanobu Baba





The MIT Press

This file has been authorized and provided by the publisher, The MIT Press, as part of its ongoing efforts to make available in digital form older titles that are no longer readily available.

The file is provided for non-commercial use through the Internet Archive under a CC-BY-NC-ND 4.0 license. For more information please visit www.creativecommons.org.

Digitized by the Internet Archive
in 2018 with funding from
The Arcadia Fund

<https://archive.org/details/microprogrammab100baba>

Microprogrammable Parallel Computer

MIT Press Series in Computer Systems
Herb Schwetman, editor

Metamodeling: A Study of Approximations in Queueing Models, by Subhash Chandra Agrawal, 1985

Logic Testing and Design for Testability, by Hideo Fujiwara, 1985

Performance and Evaluation of Lisp Systems, by Richard P. Gabriel, 1985

The LOCUS Distributed System Architecture, edited by Gerald Popek and Bruce J. Walker, 1985

Analysis of Polling Systems, by Hideaki Takagi, 1986

Performance Analysis of Multiple Access Protocols, by Shuji Tasaka, 1986

Performance Models of Multiprocessor Systems, by M. Ajmone Marson, G. Balbo, and G. Conte, 1986

Microprogrammable Parallel Computer: MUNAP and Its Applications, by Takanobu Baba, 1987

~~Microprogrammable Parallel Computer~~

~~MUNAP and Its Applications~~

Takanobu Baba

The MIT Press
Cambridge, Massachusetts
London, England

PUBLISHER'S NOTE

This format is intended to reduce the cost of publishing certain works in book form and to shorten the gap between editorial preparation and final publication. The text of this book has been photographed directly from the author's prepared copy.

© 1987 Massachusetts Institute of technology

All rights reserved. No part of this book may be reproduced in any form by any electric or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Baba, Takanobu, 1947-

Microprogrammable parallel computer.

(MIT Press series in computer systems)

Bibliography: p.

Includes index.

1. MUNAP (Computer). 2. Microprogramming. 3. Parallel processing (Electronic computers). I. Title. II. Series. QA76.8.M87B33 1987 004'.35 86-21158 ISBN 0-262-02263-X

QA76.8

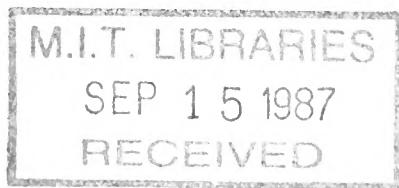
M87

B33

1987

BARKER ENGINEERING LIBRARY

Copy 3



E09365

Contents

Series Foreword ix

Preface xi

Acknowledgment xv

1 Introduction 1

 1.1 Background 1

 1.2 Project Organization and Historical Perspective 9

2 Design Principles 15

 2.1 Parallelism of Multiprocessor Units 17

 2.2 Architectural Flexibility through Two-Level, Dynamic Microprogramming 21

 2.3 Provide Orthogonal and Uniform Primitives 29

3 Basic Organization 35

 3.1 Architectural Overview 35

 3.2 Microlevel Architecture 37

 3.3 Nanolevel Architecture 50

 3.4 Interaction between Microlevel and Nanolevel 58

 3.5 Microprogram Examples 71

4 Preliminary Evaluation 79

 4.1 Application to Small Problems 79

 4.2 Evaluation of Architecture 85

5 Hardware Development 95

 5.1 Hardware Development of an Experimental Machine 95

 5.2 Procedure of Hardware Development 97

 5.3 Interface with the Other Systems 101

6 Firmware Engineering 107

- 6.1 Firmware Engineering 107
- 6.2 Description of Two-Level Microprograms 109
- 6.3 Microassembler and Nanoprogram Optimization 114
- 6.4 Optimizing Loader 128
- 6.5 Debugger 134
- 6.6 Evaluator 136

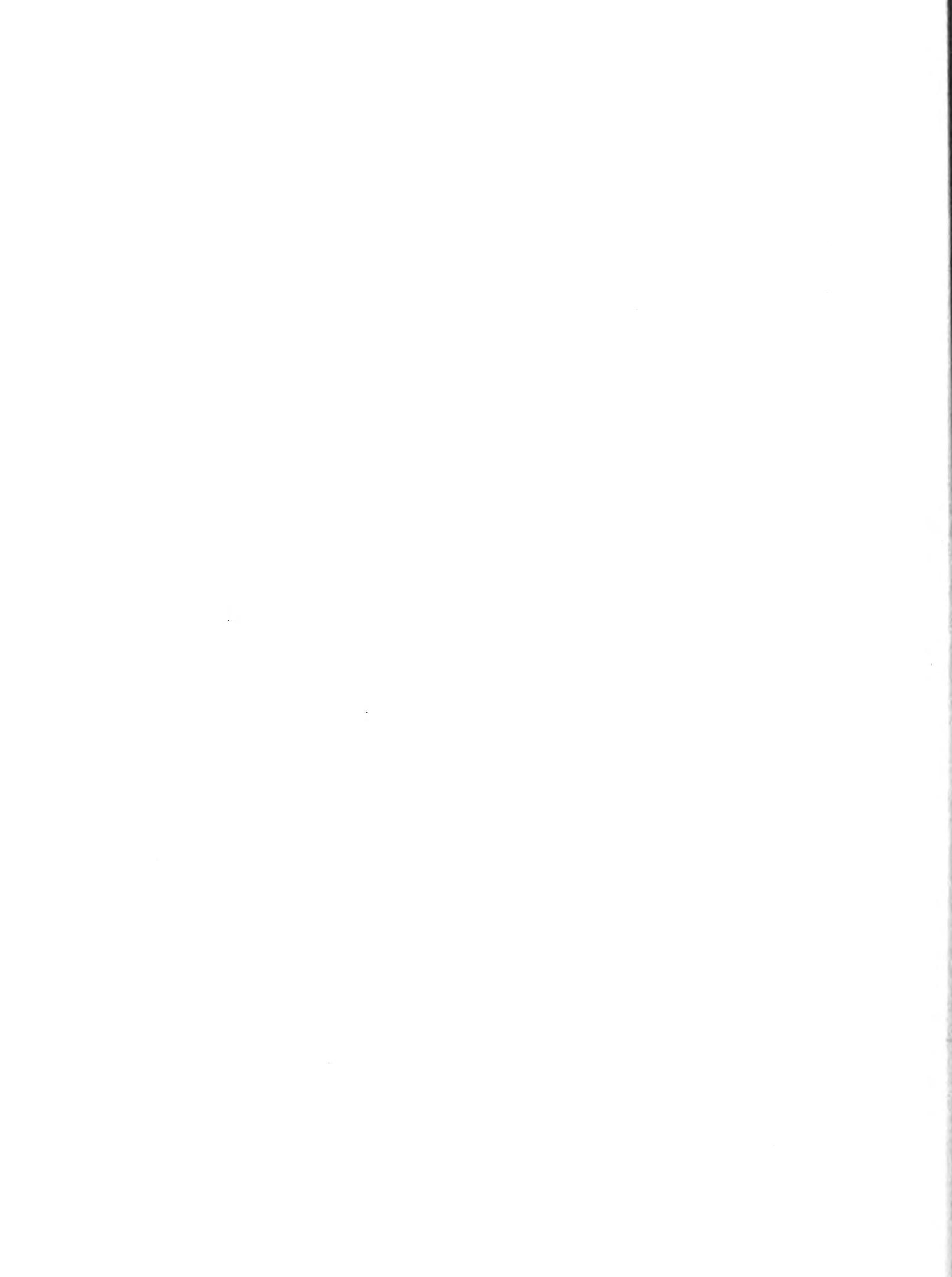
7 Applications 143

- 7.1 Basic Concepts for Application 143
- 7.2 Emulation of a Minicomputer 146
- 7.3 Tagged Architecture for a System Description Language 149
- 7.4 Architectural Support for Software Testing 163
- 7.5 Firmware Implementation of Abstract Data Types 175
- 7.6 Prolog on MUNAP 181
- 7.7 Smalltalk-80 on MUNAP 195
- 7.8 Three-Dimensional Color Graphics 210
- 7.9 Numerical Computation for Large Amounts of Data 216

8 Architectural Evaluation and Improvement 233

- 8.1 Evaluation as a Universal Host Computer 233
- 8.2 Parallelism of Multiple Processor Units 234
- 8.3 Nonnumeric Functions 239
- 8.4 Two-Level Microprogramming 243
- 8.5 Systematic Approach to Architectural Improvement 248

9 Future Directions	255
9.1 Parallelism of MUNAP	255
9.2 Application Areas and Microfunctions	259
9.3 Control Structures	262
9.4 Development and Evaluation of Two-Level Microprograms	266
9.5 Universal, Special-Purpose, and General-Purpose	268
9.6 MUNAP as a VLSI Architecture	269
 References	277
 Index	285



Series Foreword

This series is devoted to all aspects of computer systems. This means that subjects ranging from circuit components and microprocessors to architecture to supercomputers and systems programming will be appropriate. Analysis of systems will be important as well. System theories are developing, theories that permit deeper understanding of complex interrelationships and their effects on performance, reliability, and usefulness.

We expect to offer books that not only develop new material but also describe projects and systems. In addition to understanding concepts, we need to benefit from the decision making that goes into actual development projects; selection from various alternatives can be crucial to success. We are soliciting contributions in which several aspects of systems are classified and compared. A better understanding of both the similarities and the differences found in systems is needed.

It is an exciting time in the area of computer systems. New technologies mean that architectures that were at one time interesting but not feasible are now feasible. Better software engineering means that we can consider several software alternatives, instead of "more of the same old thing," in terms of operating systems and system software. Faster and cheaper communications mean that intercomponent distances are less important. We hope that this series contributes to this excitement in the area of computer systems by chronicling past achievements and publicizing new concepts. The format allows

publication of lengthy presentations that are of interest to a select readership.

Herb Schwetman

Preface

Since the advent of the computer we have witnessed many advances both in software and in hardware. In the field of software we now have various programming languages, compilers, and operating systems. In the field of hardware, VLSI (Very Large Scale Integrated) and new storage technologies have been developed; in the near future it will be possible to fabricate chips containing millions of transistors.

It is widely recognized that these advances in computer systems open up new areas for application -- and the advances in application in turn require more powerful computing systems. This rapidly growing demand can only be met by a radical change in computer architecture, where software and hardware are combined to provide powerful capabilities through parallel computation and thus flexibility for meeting a wide variety of user requirements.

This motivated us to design a new machine, called MUNAP, to show the effectiveness of low level parallel yet flexible architecture in various fields. After careful study the key concept of the machine was defined as a two-level microprogramming technology coupled with register transfer level, multiple instruction multiple data stream (MIMD) parallelism. The microprogramming was expected to enable us to tailor the machine to a wide spectrum of applications. The low level MIMD parallelism was expected to provide powerful computation

capability; it also suits the register level control by microprogramming. In the basic scheme, various primitive microoperations were provided mainly for nonnumeric processing.

This book describes all aspects of the innovative architecture in the following sequence:

Architecture design: After a short introduction in chapter 1, chapter 2 takes up several design issues, such as network and memory structures for MIMD parallelism, control structures for register transfer level parallelism, and selection of primitive operations for nonnumeric processing. Chapter 3 describes the architecture of a prototype machine. Chapter 4 evaluates the architecture, using microprogrammable parallel computer models. As they represent existing commercially available and experimental machines, this serves as a survey of machines with similar architecture.

Development: Chapters 5 and 6 describe the development of a prototype machine and supporting systems. Emphasis is placed on how we developed a large number of efficient microprograms for an MIMD parallel machine.

Applications: Chapter 7 describes the results of application in various fields. They include crucial issues in current computer architecture research: emulation, tagged architectures, language processing (such as a system description language, Prolog, and

Smalltalk-80), software testing, database systems, three-dimensional color graphics, and numerical computation. Each section includes the background of the subject, our approach to MUNAP, and experimental results. Throughout the descriptions, I have tried to clarify the effect of the architectural features.

Evaluation: Based on the application, chapter 8 evaluates the architecture. It also proposes and applies a systematic method for improving architecture. The method is so general that it can be applied to any other architecture. Chapter 9 discusses possible applications and alternatives of the architecture, considering the requirements from the VLSI viewpoint.

This book has two purposes: to introduce the MUNAP architecture and the results of its application to computer professionals in general, and for use as a textbook for people who want to design and develop a machine with innovative architecture. The reader is expected to have a good grasp of computer system fundamentals. In particular, the reader should be knowledgeable about computer system organizations, have an understanding of programming language processing, and have a grasp of the concept of microprogramming.

It is worth noting that the MUNAP architecture has a strong relationship to such recent architectural innovations as the very long instruction word (VLIW) architecture and the reduced instruction set computer (RISC) because it consists of two types

of simple instruction sets, the first level of instruction controlling multiple instructions at the second level. This can be viewed as a single VLIW architecture or a complex of RISC computers.

We believe that this book presents comprehensive research directions in the area of computer architecture, and we hope that it will encourage the people planning to develop a "next generation" computing system.

Acknowledgment

I gratefully acknowledge the help of many people who have contributed ideas, hard work, and enthusiasm to the MUNAP project.

At Utsunomiya University, my colleagues, Kenzo Okuda, Katsuhiro Yamazaki, and Ken Ishikawa, have led several research projects related to MUNAP, which enabled me to write this book. I am also grateful for their invaluable comments on the draft of this book.

A group of undergraduate and graduate students helped with the implementation: Hiroyuki Kobayashi, Junichi Hiroki, Yasuhisa Fukuda, Youichi Katahira, Masato Nakai, Hitoshi Yamamoto, Keiji Tanabe, and Ichiro Watanabe for architecture design and hardware development; Akihiro Saito, Hajime Kajioka, Yoshihide Morioka, and Nobuyuki Hashimoto for the development of the microprogramming system; Tadashi Katayama, Kaoru Kiriyama, and Mitsuru Ikeda for the development of the evaluator and optimizing loader; Hiroyuki Kanai, Kazuhiko Hashimoto, and Yasuko Ono for the development of the MUNAP system description language; Takumi Ohtani, Norihiro Sato, and Kumiko Iwata for the MUNAP abstract data type database machine; Toshiyuki Sano and Tomoko Sakurai for the MUNAP software testing system; Shinji Suzuki and Tsutomu Yoshinaga for MUNAP graphics; Hideru Doi and Shousaku Ookawa for MUNAP Smalltalk-80; Masayuki Inagawa, Hideki Saito, and Tomoyuki Kase for MUNAP Prolog; Toshinobu Ooi and Hiroyuki

Sasaki for the MUNAP emulator; and Hideo Shibaoka and Yoshiji Takahashi for MUNAP numerical computation. Some of the projects were partially supported by the Japanese Ministry of Education, Science and Culture.

Several researchers provided valuable suggestions on the architecture and development. Hiroshi Hagiwara, Shinji Tomita, Kiyoshi Shibayama, and Haruo Niimi at Kyoto University gave us invaluable comments based on their pioneer work in Japan. I thank Katsuya Hakozaki of the NEC C&C Systems Research Laboratory, Tadaaki Bandoh of the Hitachi Research Laboratory, and Shigeru Oyanagi of the Toshiba Central Research Laboratory for their comments on the architecture at its early stage of development.

Thanks are due to Frank P. Satlow, executive editor of The MIT Press. Without his suggestions and encouragement, I would not have tried to write a book on MUNAP in English. Jack Dennis at MIT gave me invaluable advice on the outline of the book. At our laboratory Morihito Watanabe, Takao Iwasaki, Naoki Saitou, and Tohru Iijima helped me to prepare the camera ready manuscript.

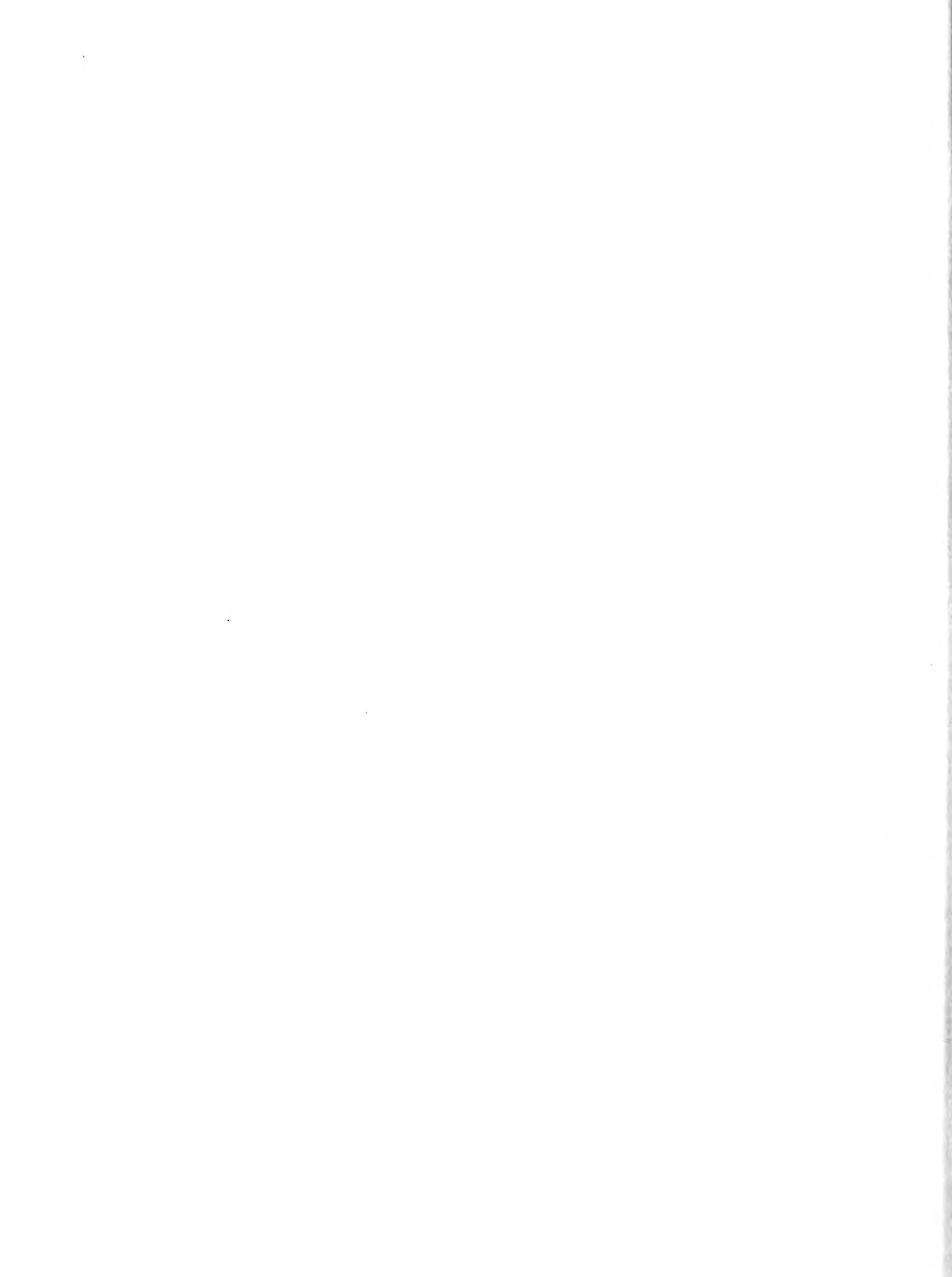
I would like to thank several academic societies, in particular, IEEE, ACM, IFIP, and IECE Japan, who published the papers on MUNAP used as the basis of descriptions in this book, which are included in the references. AFIPS and IEEE permitted reproduction of figures 4.1 and 4.3 from their publications.

Finally, I must thank my wife, Hideko, and my children, Sayako, Chikako, and Hironobu, for their moral support.

Trademark Notice

Connection Machine is a trademark of Thinking Machines Corporation. ECLIPSE is a trademark of Data General Corporation. MC68000 is a trademark of Motorola Corporation. MULTIBUS is a trademark of INTEL Corporation. MUNAP is not a trademark. Smalltalk-80 is a trademark of Xerox Corporation. UNIX is a trademark of American Telephone and Telegraph Company.

Microprogrammable Parallel Computer



1

Introduction

1.1 Background

Since there may be readers who are not familiar with computer architecture and microprogramming, the first order of business is to introduce the background to the research and define precisely the terms used in this book. This is the basis of not only computer architecture but also the architectural features of our machine.

1.1.1 Microprogramming

A digital computer can be partitioned into five distinct functional sections: input, memory, arithmetic and logic unit (ALU), control, and output. These five sections communicate with each other through electronic signals that represent data,

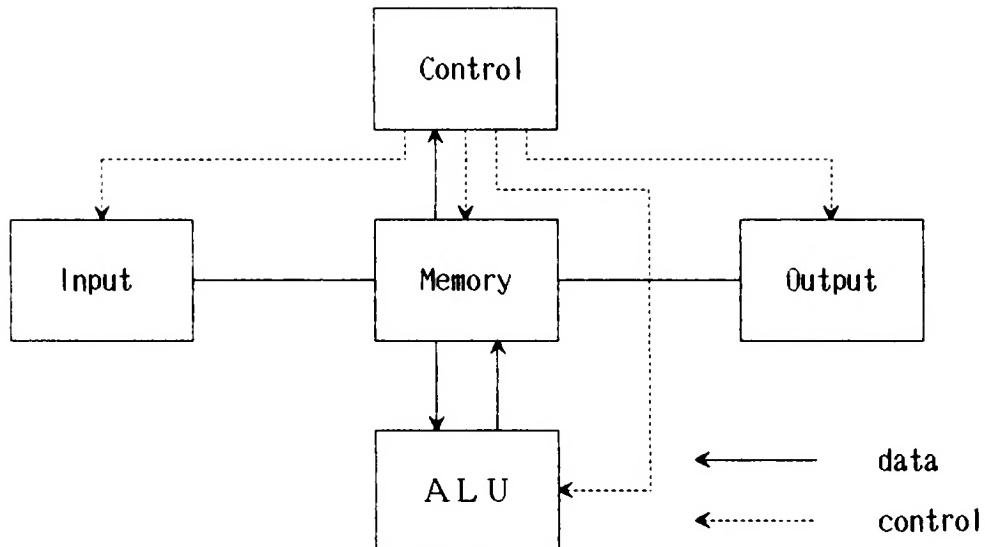


Figure 1.1
Functional organization of a digital computer.

instructions, and control signals, as shown in figure 1.1. The control section issues control signals and effects the order, timing, and direction in which the data and instructions flow. It fetches and decodes an instruction from a sequence of machine instructions in the memory; the results of decoding specify data movements, ALU operations, and/or the next instruction address. There are two methods for implementing the control section. In the so-called *hardwired* computer, the control element is a sequential logic circuit that sequentially executes the primitive operations that effect a machine language instruction. Microprogramming is an alternative method of implementation that can reduce the complexity and inflexibility of the control section. In the so-called *microprogrammed* computer, the control logic is represented as a sequence of primitive operations, such as opening a gate and an elementary ALU operation. The

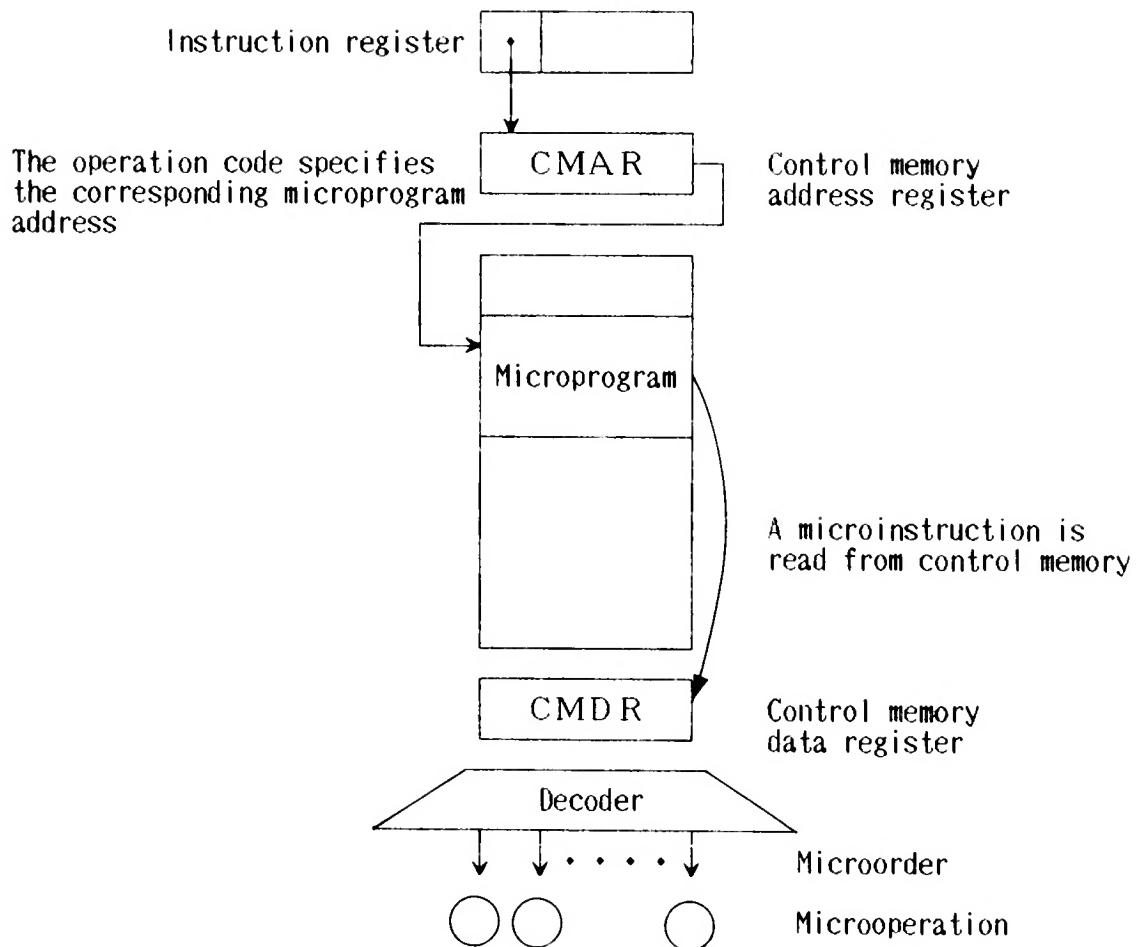


Figure 1.2
Execution of microoperations on a microprogrammed machine.

primitive operation is called **microoperation**. The **microorders**, which specify corresponding microoperations, are encoded to form an instruction, called **microinstructions**. Thus, a complex operation for a machine instruction can be represented as a sequence of stored microinstructions in a memory unit, called a **control memory** or **microprogram memory**. In a microprogrammed computer, the execution time for one microinstruction is the basic unit of the timing control, and is called a **microcycle** or a **machine cycle**. Figure 1.2 shows the relationship between these concepts. This method of control is similar to writing a program

in which a given arithmetic or logic operation is executed by giving the CPU (Central Processing Unit) of the system a step-by-step description of the job to be done. This program, then, is a series of subcommands for the microscopic functions built into the system, and thus, the word *microprogramming* was coined.

The first memory technology developments that influenced microprogramming were a variety of types of read only memory (ROM). The microprograms stored in the ROM were used to perform machine language instructions. This property was utilized to realize the machine instruction set of an old machine as a microprogram on a new machine and thus keep the compatibility. This technique was called *emulation*. In this era the term *firmware* was coined by A. Opler to describe "microprograms resident in the computer's control memory specializing the logic design" [OPLE67].

The development of fast read/write semiconductor memories has provided new capabilities in implementing machines by microprogramming. Many computers use such memories as control memories, and some computers permit users to access and modify the memory. This technique is called *dynamic microprogramming*. It provides a great deal of flexibility to computer architectures. For example, a dynamically microprogrammable machine can be microprogrammed to emulate any (including virtual) desired target machine. A machine with this capability is called a *universal host*. The notion of host machines being microprogrammed to emulate target machines is perhaps too restrictive, especially if the target machine is perceived as a

machine language programmable computer. Indeed, a great deal of work has already been done in microprogramming machines for the direct execution of higher level languages, microprogramming of operating systems, and even direct microprogramming of applications programs. The reader will understand that the applications of our machine can be conceived as an effort to realize a universal host machine, using a flexible yet powerful architecture.

1.1.2 Parallel Processing

An important trend in the design of computing systems is toward the greater use of parallel processing, i.e., more than one facility performing tasks in parallel. It is true that the technological advances of integrated circuits and mounting techniques allow us to construct a fast machine. However, the computation speed improvement obtained by them has an upper limit, the speed of "light," because the signals are transmitted through electrical circuits. Thus, at a given component speed, the computation rate in a stored program computer can be increased only by increasing the concurrency of operations.

The parallelism is categorized from two viewpoints: type and level. The first viewpoint, type, divides the parallelism into three classes by the multiplicity of the instructions and the data stream. The first category, SIMD, means Single Instruction, Multiple Data stream; here uniform data are usually processed by

6 CHAPTER 1

multiple processors in the same manner under the control of a single instruction stream. Examples of this class of machine include ILLIAC IV, BSP (Burroughs Scientific Processor), PEPE (Parallel Element Processing Ensemble), MPP (Massively Parallel Processor) [POTT85], and PAX (Processor Array eXperiment) [HOSH83]. The second category, MISD (Multiple Instruction, Single Data), is interpreted as a pipeline processing, where a single data stream receives multiple functional operations while it flows through a pipe. There are many commercially available pipeline processors, such as CDC STAR 100, and CDC Cyber 205. The third category, MIMD, represents a Multiple Instruction, Multiple Data stream, where mutually independent instruction streams control their data streams. C.mmp and Cm* are famous examples of this category developed at Carnegie-Mellon University. The HEP computer is an MIMD computer consisting of one or more pipelined MIMD process execution modules that share memory [KOWA85]. Generally speaking, the SIMD systems are meant for limited application, such as array processing. If we want a high performance system that also allows general purpose computing, we must use an MIMD system.

The second point, the level of parallelism, may be divided into several classes: circuit, the LSI (Large Scale Integrated) chip, register transfer, processor unit, and computer. The difference here between the processor unit and the computer is that the processor unit is supposed to be a component of a tightly coupled multiprocessor computer. On the other hand, the computers may be connected through a channel to make up a loosely

coupled computing system. We can categorize a given computing system according to these points. Notice, however, that there may be several ways of categorizing by application.

1.1.3 Computer Architecture and Software/Firmware/Hardware Trade-Offs

The computer architecture can be defined as the distribution of function across a given level or boundary of a computing system and the precise definition of the boundary [MYER79]. That is, if we are establishing the architecture of a certain level of the system, the first step is to determine which of the system's functions will be performed above the level and which will be performed below the level. Once this has been accomplished, the second step is to develop a precise definition of the level's interfaces.

This implies that there are distinct layers of architecture within a computing system. One way to define computer architecture is to decompose the system's functions into three layers: *software*, *firmware*, and *hardware*. The dividing point between the system's software and its firmware is the machine instruction interface. It represents the abstraction of the system's physical representation as seen by its software. Distributing function above and below this level is the process of *computer architecture* in a narrow sense. It is the definition of the conceptual structure and functional behavior of a machine as seen by a machine language programmer or a compiler writer; it

ignores such factors as the processor's underlying data flow and controls, logic design, and circuit technology. In this book, we use the term computer architecture in a broad sense that includes all aspects of the design of the interfaces and underlying hardware. In a microprogrammed computer, the microinstruction set defines a lower dividing point between the firmware and its underlying hardware. This type of architecture is called *microarchitecture*. Notice that, in a hardwired computer, the machine instruction set is directly implemented in hardware as described above, and there does not exist a microarchitecture. To distinguish a machine language architecture from the microarchitecture, we use *macroarchitecture* in this book.

The computer architect, then, makes the decisions on the two major interfaces: macro- and microarchitectures. It is important for the architect to be concerned with the efficiency of problem solution, rather than the average raw speed of the machine instructions; that is, the architect must consider efficiency from the viewpoint of the programming language/ compiler/machine triplet. It is not the raw instruction speed that makes for an efficient system; rather, the "semantics" of the instruction set (the amount of function performed by the machine) has a more significant effect on performance. This implies that if the semantics of the macroarchitecture is not defined properly, it will make the job of the compiler writer very complicated. The phenomenon is known as the *semantic gap*, which is a measure of the difference between the concepts in a high level interface and those in a lower one. It is pointed out that most current

systems have an undesirably large semantic gap in that the objects and operations reflected in their architectures are rarely closely related to the objects and operations provided in the programming languages [MYER78]. This gap contributes to software unreliability, performance problems, excessive program size, compiler complexity, and distortions of the programming languages, all of which contribute negatively to the performance of data processing.

There are several attempts to bridge the gap [BERN81]. The most active research and implementations are in the field of high level language processing. The language oriented architectures are called *high level language machines* and they reflect the language constructs (procedure calls and so on). The other attempts include *array processors* and *vector processors* for multiple data processing, *database machines* for database processing, *graphics processors* for two- or three-dimensional data processing, *signal processors* for signal data processing, and so on. These machines may be viewed as the results of the efforts to narrow the gap in each of the fields.

1.2 Project Organization and Historical Perspective

The MUNAP project has the following three stages [BABA83b]:

1.2.1 Architecture Design and Hardware Development

The project started by defining the design objectives of the machine. According to the objectives, the key idea defined was a two-level microprogramming scheme coupled with a register

transfer level parallelism. The hardware organization and the associated micro- and nanoinstruction formats (so-called microarchitecture) were defined. The architecture design was done by a professor and an undergraduate student in half a year. This small number of workers enabled us to design a machine with well-defined principles. The detailed design of hardware and the development of a prototype of a processor unit was done the next year, mainly by a group of five persons. After checking the logic of the processor unit with that of two-level control scheme, we extended the number of processors to four, using the same wiring list as in the first prototype processor. This part is one of the most successful aspects of the hardware development of multiprocessor computer.

1.2.2 Support Software Systems Development

Our initial experiences with MUNAP led us to feel that some good support software systems are necessary for utilizing MUNAP. The first attempt was a *microprogram debugger*, developed on the console processor. It allows users to do console operations, such as general reset, load/store microprograms, start/stop the execution, read/write the facility values, and step run the microprograms for debugging. The next support system was a *microassembler*, which allows the user to describe microprograms at register transfer level. For example, the register-register transfer may be described by an assignment statement. As the size of microprograms grew, a *relocatable microprogram loader* became necessary and was developed. The loader receives object

microprograms of the microassembler and relocates them to produce a load microprogram module. An *evaluator* has also been developed to get data while running the machine. The evaluator monitors the stepwise execution and produces the data for evaluating dynamic properties of the machine, such as microinstruction usages.

1.2.3 Application to Research Problems

Using the basic hardware and software systems, attempts to apply the computer to a wide variety of applications have been made as follows to clarify the effect of the architectural features:

1. Emulation: Historically, emulation is a basic application of a microprogrammed computer. A minicomputer at our laboratory was selected as a target machine for which microprograms have been developed to emulate the machine.

2. Tagged architectures: Two tagged architectures were designed and developed. The first one is for a system description language of MUNAP called MSDL. The language has a C-like syntax, reflects the architectural features of the machine, and is realized using a tagged data structure. For example, the tag indicates data type, capacity, undefined, number of references, and so on. Through experimentation the tag is shown to be effective not only for enhancing the user interface but also for checking erroneous actions caused by incorrect data. The second tagged architecture supports a software testing system for a low level linked list language. This system provides a set of

commands to test programs interactively. For example, the user can not only specify the type and range of variables but also get information about frequency of the execution of statements.

3. Abstract data types for database processing: An abstract data type is a class of objects that is completely characterized by a representation-free specification of the operations available on those objects. In this scheme a specification and the representation are separated. This concept is very useful for realizing the data independence and security of database systems. The complete specification defines constraints for the objects, and it could contribute to database semantics. We defined necessary objects for a database system and the operations available on them. Utilizing the low level parallelism and flexibility of the MUNAP architecture, we realized the operations through firmware. The results have shown that the hardware and firmware implementation attain the data encapsulation of a database without performance problems as pointed out for software implementation.

4. Prolog and Smalltalk-80: Computer architects have had to cope with the advent of new programming languages based on innovative ideas. Remarkable languages among them are a logic programming language and an object oriented language. We selected the most popular languages from them as the target high level languages: a logic programming language (Prolog) and an object oriented language (Smalltalk-80). The major idea here is to utilize our computers' low level parallelism and flexibility for improving

the performance of the language processors. In particular, the unification parallelism for Prolog and the intramessage passing parallelism for Smalltalk-80 have been investigated.

5. Three-dimensional color graphics system: Computer graphics is an extremely effective medium for communication between man and computer, producing images whose appearance and motion make them quite unlike any other form of computer output. One of the most important problems in the field is that the system requires the computation of large amounts of data at high speed to show the results in real time. In particular, three-dimensional color computer graphics requires powerful computational capability in order to provide realistic images, such as hidden surfaces and shadowing. We applied the MUNAP computer to this problem to clarify the effect on it of parallelism.

6. Numerical computation: As a universal host machine, MUNAP was expected to provide an effective tool in the area of numerical computation. This aspect of application includes such important problems in the field as sparse matrix computation and Fast Fourier Transformation (FFT).

Actually, these stages did not progress serially. The development of microassembly language started just after defining the microarchitecture, in parallel with the development of hardware. The application part of the last item has been performed in parallel with the development of system description language.



2

Design Principles

Our motivation in design and development was to prove the effectiveness of innovative computer architecture that is independent of implementation techniques. Thus, it was very important to make the design objectives clear and have well-defined design principles based on these objectives. They were supposed to guide the architect so that the architecture should appear to have been conceived, designed, and documented by a single mind. The design objectives of our machine are as follows:

1. The architecture should be innovative enough to show new possible directions beyond von Neumann architectures, even after a lengthy development.
2. In a research oriented environment, the machine should

provide a flexible and powerful tool to investigate a wide spectrum of research areas.

These objectives suggest as a direction a universal host machine that is reasonably amenable to various application fields, such as emulation, language processing, and numerical computation, as mentioned in chapter 1. The keys to realizing a good universal host (and, we believe, they are also the keys to a commercially available machine) are

- o *flexibility* for meeting a wide variety of environments,
- o *concurrency* of the hardware operations for utilizing the inherent parallelism of a given environment, and
- o the architectural *uniformity* and *orthogonality*.

We considered possible architectures according to these principles. In particular, our attention was attracted by the *flexibility* of a two-level microprogramming scheme combined with the *parallelism* of multiprocessor units [BABA80]. These features are quite an attractive way to implement a powerful universal host computer to be utilized in a wide spectrum of areas. Further, we decided to enrich nonnumeric functions in our machine as to the uniformity and orthogonality of elementary hardware functions. Before going into detailed descriptions of the architecture, let us discuss the principles a bit more.

2.1 Parallelism of Multiprocessor Units

2.1.1 MIMD Register Transfer Level Parallelism

As we mentioned in 1.1, there are several types and levels of parallelism. We decided to adopt an MIMD type, register transfer level one. The selection for the low level parallelism results naturally from the principle of utilization of microprogramming, which controls hardware at the register transfer level (RTL). The next selection is the parallelism of the MIMD type (multiple instruction, multiple data streams) because this type is most comprehensive, covering both the SISD and SIMD types. It allows us to construct a high performance, general purpose computing system applicable to a wide range of research problems. If we assume the same facilities to be controlled, the machine of SIMD type is much easier to develop. However, it limits the areas of application. Having the MIMD parallelism, the computer can easily exploit the inherent parallelism of a given environment to the utmost. On the other hand, most of the conventional parallel and pipeline processors utilize only the explicit parallelism between a large number of uniform data; the typical example is array processing. Thus, our register transfer level MIMD architecture is to be aimed at not only utilizing an explicit parallelism for a large number of uniform data but also exploiting an implicit parallelism from a seemingly sequential algorithm.

2.1.2 Fast Execution of Mutually Dependent Operations

In order to execute programs in multiprocessor units in parallel, the data on the processors should be mutually independent. However, in a situation where a single task is decomposed into several subtasks, it is unrealistic to expect that the data are completely independent and that the subtasks can continue their execution without an interaction. This means that the result of a computation in a processor is sometimes required as an input to another processor, which indicates the importance of the high speed execution of a series of mutually dependent operations. If a computer has such a capability, it can exploit a maximum parallelism by performing mutually independent operations in parallel, and mutually dependent operations in series at a high speed. We regarded the network structure as the key to the implementation of such an operation. A shuffle exchange network was chosen to realize the operation (see below).

2.1.3 Parallel Architecture and Design Parameters

To utilize fully an MIMD register transfer level parallelism, the basic structure of our machine was made a tightly coupled multiprocessor organization, where multiple processors are connected to multiple banks of main memory through a shuffle exchange network. This structure allows a flexible data transfer between multiple processors and main memory banks.

Processor units: The number of processor units is a basic design

parameter. The network structure usually allows an extension of processors by multiples of two. Needless to say, the greater the number of multiprocessor units, the larger the possibility of parallelism. However, it is very hard to describe microprograms of a computer with eight or more independent processor units for each microcycle. Moreover, the effect of parallelism by more than seven or eight processor units may be uncertain under a sequential control scheme [MOT079], provided the problem does not have an explicit parallelism. On the other hand, the two-processor unit organization does not seem to be attractive because of its limited parallelism. Thus, we decided to provisionally adopt four-processor units organization with 16-bit word length. However, this does not mean our architecture is limited to four processor units. The number may be changed, depending on the environment.

In a processor unit, a large capacity scratchpad memory is provided to support local processing without communicating with the other processors and accessing main memory for a relatively long period. For communication outside a processor, each processor contains port registers for inputting and outputting data. As we shall see later, these may act as a critical region between the outside and the inside of a processor unit for exchanging data asynchronously.

Main memory: The main memory is interleaved to allow the multiple processors to access the contents in parallel and thus avoid memory access bottleneck. We selected a byte as the unit of

interleave so that the user can define his own data structure at the unit of the byte. Data units smaller than a byte are to be covered by providing a variety of nonnumeric operations in network and processor units. Thus, eight banks of 1-byte memory were determined to be the basic memory structure.

Shuffle exchange network: Various network structures have been designed to facilitate parallel data permutation [FENG81]. The key factor in determining a suitable architectural structure is the network topology, which views a network as a graph in which nodes represent switching points and edges represent communications links. From the viewpoints of topology, various network structures may be classified from the simple (linear array and ring structure) to the complex (completely connected structure). Among them, we selected a *shuffle exchange network* because of its simple yet comprehensive structure [LAND76]. It not only performs regular transformations, such as shift and mirror transformation, but also covers some irregular transformations. We added a broadcasting function to each cell to enhance the applicability of the network. Theoretically, n input data item need $\log_2 n$ stages of n cells for data exchanging and broadcasting. Thus, if we divide the input into 64 1-bit, 32 2-bit, 16 4-bit, and 8 8-bit, then 384 ($=64 \log_2 64$), 160, 64, and 24 cells are required, respectively. Our selection of the third configuration (16 4-bit) is the compromise between the number of the cells and functional redundancy between the network function and the processor function. The 32 2-bit configuration would require 2.5 times as many control bits as the 16 4-bit one.

Further, as a processor unit function is to be enriched to include various data handling operations in a 1-bit unit, the 1- and 2-bit operations have much redundancy. On the other hand, if we employed 8 8-bit, the gap between processor unit 1-bit data processing functions and the network 8-bit data permutations would be intolerable for the users. The network provides the facilities for not only data transfer between multiple processor units and multiple main memory banks but also fast execution of serial operations among processor units.

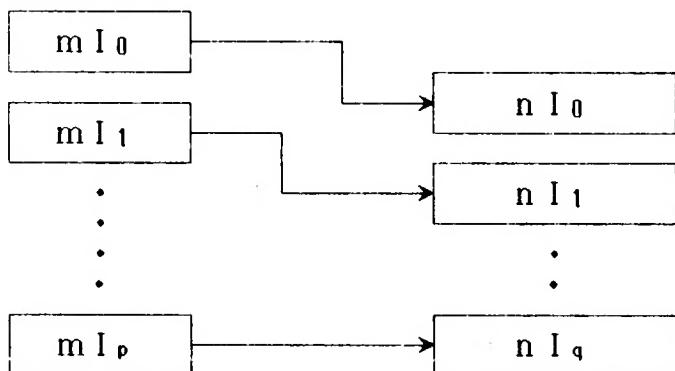
2.2 Architectural Flexibility through Two-Level, Dynamic Microprogramming

2.2.1 Two-Level Microprogrammed Multiprocessor Architecture

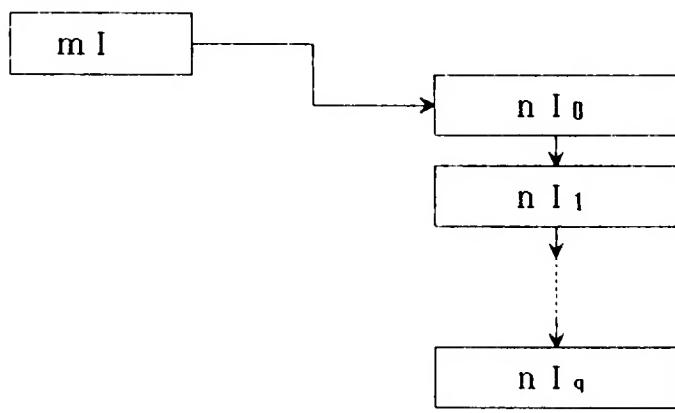
Dynamic microprogramming is a basic technology for implementing a flexible architecture because by this means a machine can change its architecture by reloading microprograms for a given environment. They may be microprograms for any target, such as the machine instruction set of a computer, a high level language, and a numerical computation problem. In particular, a *two-level microprogramming scheme* attracted our attention because of its very flexible control structure. In this scheme, the control memory is divided into two parts, namely, microprogram memory and nanoprogram memory, and the combination of the microprogram and nanoprogram may be defined by firmware level users.

There are two types of control store structures, as

illustrated in **figure 2.1**. In the type shown in figure 2.1a, the microinstructions (**mIs**) in the first level memory are narrow, consisting primarily of pointers to nanoinstructions (**nIs**) in the lower level nanoprogram memory. They also contain information about branching in the microsequence and, in some cases, literal values to be assigned to specified registers. The nanoinstructions are wide, providing fairly direct, decoded control of the execution unit. This scheme requires fewer bits



(a)



(b)

Figure 2.1
Two-Level Microprogramming: (a) nanoaddress specification
by micro; (b) interpretation of micro by nano.

of memory than a standard organization when short microinstructions that move data are executed frequently, and when several short microinstructions refer to the same long nanoinstruction. Examples of this architecture include the Burroughs Interpreter [REIG72] and the Motorola MC68000 [STRI78]. In the second type, shown in figure 2.1b, the nanoinstructions in the lower level storage unit interpret microinstructions in the upper level storage unit, in much the same way as ordinary microinstructions interpret machine language instructions. This scheme permits flexibility in the design of upper level microinstructions as well as in the design of machine language instructions. The Nanodata QM-1 is an example of this type [ROSI72].

Generally, the upper level control store contains narrow, encoded microinstructions, which represent a set of microoperations with a small, limited parallelism; this type of microinstructions is called a *vertical microinstruction*. On the other hand, the lower level control store contains microinstructions with many more bit lengths and hence can exercise more control over machine resources. In contrast to vertical microinstructions they are called *horizontal microinstructions*.

Adoption of a two-level microprogramming scheme coupled with the utilization of register transfer level parallelism lead us to the quite interesting architecture shown in figure 2.2. In this architecture a microinstruction in the microprogram memory specifies the nanoprogram addresses of several tightly coupled

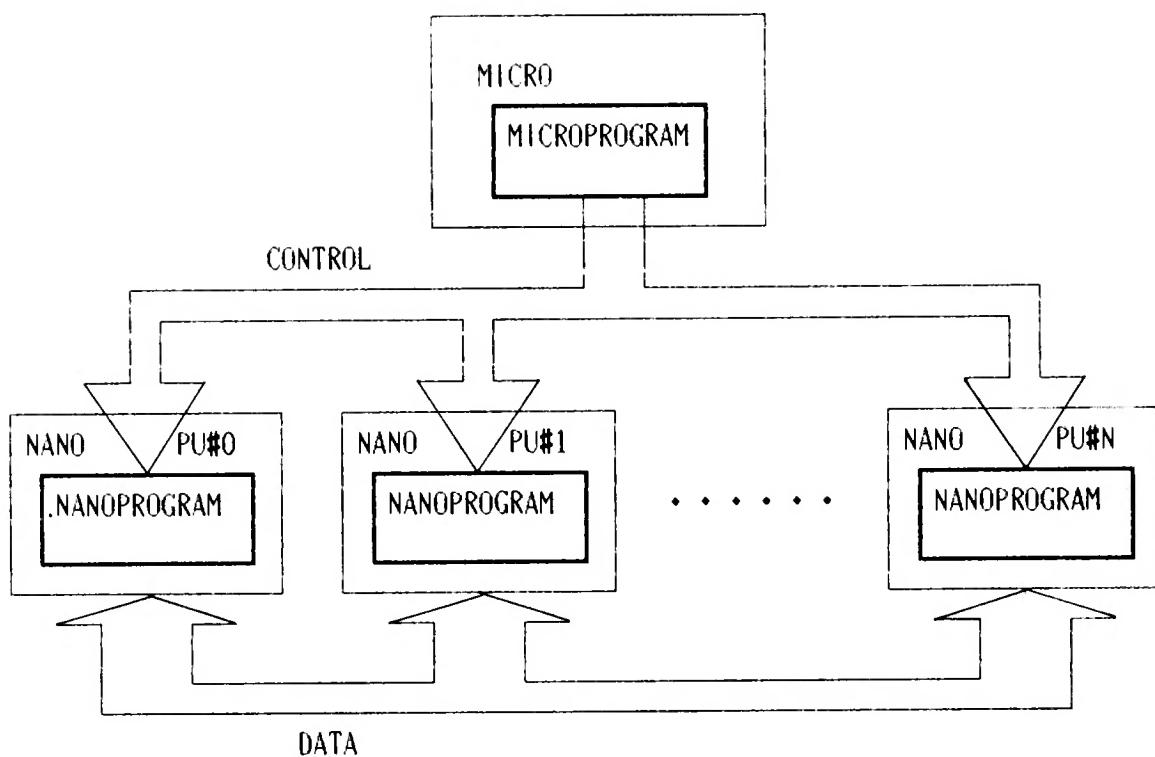


Figure 2.2
Conceptual view of a two-level microprogrammed multiprocessor architecture.

processing units. The nanoprogram controls the operations inside the processor units (PUs). The microprogram also controls global data flow outside the processor units and coordinates the execution of multinanoprograms. The method does not precisely fall into one of the above two categories. However, conceptually, it is much like the first category. This scheme is expected to have the following effects:

1. flexibility for meeting a wide range of system requirements through writable microprogram and nanoprogram memories,
2. utilization of maximum parallelism between multiprocessor

units under single microprogram control, and

3. savings in total control storage requirements by having nanoinstructions that are common to more than one microinstruction.

The first two factors allow a single microinstruction to be applied to any combination of the processing elements for each unique application of the operation. From the viewpoint of hardware and VLSI implementation, the scheme has the following advantages over the usual single level microprogramming scheme:

4. modularity of hardware units, under the distributed control function of nanoprogams and
5. closeness between the control and controlled parts.

The fourth item will be clear because in our architecture each processor unit is to have its own nanosequencer and generate control signals inside the unit by decoding nanoinstructions. On the other hand, usual single level control should decode a microinstruction before sending the result to processing units. Sometimes, each unit may have a decoder for a microinstruction field. However, the total amount of information conveyed every microcycle from a microprogram should be much larger than is given by our scheme. It is important that these factors be advantageous for VLSI implementation, where the transfer speed of data and control signals, as well as the compaction of chip area,

rather than the computation speed, have profound implications.

Thus a two-level microprogrammed multiprocessor architecture became the key idea of our machine, named MUNAP, for **MUti-NAnoProgram** machine.

2.2.2 Job Sharing between Micro- and Nanoprograms

One of the important design decisions for the two-level control scheme is to determine the major roles of the two levels and define the interface. Included are the controls for data flow, computations, sequence, and the interaction between the two levels. The principles that direct our decisions are (1) The major role of micro is the control of microprogram sequence, which in turn activates the nanosequence. (2) Avoid functional redundancy, which means that the controlled parts of the two levels should be disjointed. The resulting decisions are the major data flow control and sequence control by micro, the PU (Processor Unit) data flow and computation by nano, and the sharing of the interaction between the two levels.

2.2.3 Micro-Nano Interaction

The micro- and multiple nanoprograms interact in several ways. They are divided into control, data transfer, and timing.

- 1. Control:** A process view of the execution of micro- and nanoprograms poses a quite interesting issue, namely, distributed control in a tightly coupled multiprocessor system. In the basic interaction, the micro activates multiple nanoprocesses and the nanos indicate their termination. The micro may wait for the end

of all the nanoprocesses. However, it can proceed to the next microinstruction execution if the processors, on which are the nanoprocesses to be activated by the microinstruction, terminate the execution of nanoprocesses activated by the previous microinstruction. Clearly, the latter scheme is more efficient than the former. We decided to include both schemes and allow the user to choose. The second interaction involves transferring the test result from nano to micro so that the micro can control its sequence based on the nano test result. The necessity of the interaction is based on the job sharing, in which the micro has the basic sequence control capability and the nano does the arithmetic and logic operations. Fast transmission of the test result was desirable. The third and the most flexible interaction concerns requests from one level to another. It was defined to allow the micro- and nanoprocesses to communicate with each other by the interrupt mechanism. There may be some who doubt the effectiveness of such a mechanism in a tightly coupled multiprocessor system; however, a later experiment on the Prolog processor proved it to be very effective for parallel unification processing in Prolog programs.

2. Data transfer: The fast execution of mutually dependent operations by multiple processors needs some mechanism that allows quick data transfer from one processor to another through a shuffle exchange network. The total transfer was divided into intra- and interprocessor transfers, which are controlled by nano and micro, respectively. Thus, the micro and nano should be

combined to control mutually dependent operations at a fast speed. In addition, various configurations of processors should be realized by a flexible network control structure, of which the simplest is serial operations of all the processors; in this case, the computation result of one processor is broadcast to the other processors.

3. **Timing:** Although the flexibility of two-level microprogramming scheme has been widely recognized, its timing overhead prevented its spread to many computers as a method for designing the control part. The scheme requires reading and decoding a control word two times. Thus, analysis of the timing is the key to making it acceptable. We designed the two levels of operations for control words to overlap as well as possible.

2.2.4 Functional Control

This control method uses functional control registers, sometimes called *setup registers*, the values of which indicate the microoperation to be performed by a functional unit [AGRA76]. In a situation where functional units perform similar operations and where their differences may be controlled by the functional register, this scheme provides substantial execution step saving. We decided to adopt this scheme basically for all the functional units. Thus, the user can control the operation of a functional unit directly by a control word or functionally by a functional register.

2.3 Provide Orthogonal and Uniform Primitives

Our next concern is how to design a more detailed architecture in accordance with the above. We needed clear design principles that would have a significant effect on system quality. Several researchers have discussed the principles for computer architecture [WULF81]; we feel they are quite similar to those we had in mind when designing our machine.

2.3.1 Provide Primitives

The "Provide primitives" principle, the natural consequence of employing a microprogrammable architecture, is to design a machine as a collection of mutually independent, primitive operations and provide a tool for combining them arbitrarily through microprogramming. There are two major advantages of this approach; the first one concerns the software, and the second the hardware. The first advantage is based on the fact that one finds many different views of essentially similar concepts among the common higher level architectures. For example, the control structures of PASCAL cannot be coded directly in FORTRAN. Features such as COMMON and EQUIVALENCE are not present in PASCAL. The functional requirements for the database processing are different from those for high level language processing. A machine that builds in instructions satisfying one set of these requirements cannot support other languages. A machine that attempts to support all the requirements will probably fail to support any of them efficiently. An ideal solution to this problem would provide primitive operations that could be

recomposed to model more closely the features actually needed by tailoring the instruction set to the needs of a particular environment. The second advantage is based on the fact that designing irregular structures is very expensive. The replication of a simple structure not only decreases the cost of design and implementation but also eases machine testing, which is one of the most important aspects of enhancing hardware reliability. The set of primitive operations will naturally be realized by a collection of small hardware modules with a simple structure.

2.3.2 General Principles for Defining Primitives

For the general principles that guide our function definition process, we may quote from other papers [MYER78, WULF81]; however, the definitions are ours and are slightly different from the meanings of those in the papers.

Orthogonality: The primitive fuctions should be mutually independent. To avoid redundancy, functions should not have similar meanings.

Uniformity: All function definitions should be consistent.

Completeness: There should not be a case where a required operation cannot be composed from the primitives. All the possible operations should be covered by the primitives.

Orthogonality excludes the existence of similar functions. It not only avoids redundancy but also simplifies the

micropogramming process because a required function may always be realized in one way. Uniformity requires consistent definitions of primitives. For example, if an ADD operation may input from certain resources, the other arithmetic and logic operations should input the same ones. If a functional control register is attached to a unit in some way, similar units should have functional control registers in the same way. Completeness requires the inclusion of all the necessary primitives for intended applications. This proves that functions at the same level may be realized at the same cost. However, unpredictable operations may be required after the development of an experimental computer. For this case, it is best to have an optimal capability of extension. For example, some microinstructions, fields, and codes may be left undefined.

2.3.3 Primitives Selection for Nonnumeric Processing

To select primitives, we surveyed the functional requirements from the perspective of possible areas of applications and defined the primitive functions that may be used to recompose the required functions in a given environment. Besides the usual operations we defined the following functions, mainly for nonnumeric processing:

Bit processing: Bit processing capabilities are required in various situations. An example is the implementation of a tagged architecture, where each data item has the tag bits that identify the attributes and give some meaning to the data. In order to

handle the tag, we need bit operations, such as bit set and test. Several high level programming languages provide explicit bit processing functions, which will be effectively implemented if the underlying hardware provides primitive bit operations. The bit vector approach of the LEECH database machine shows the feasibility of a machine in which bit processing plays an important role in such operations as joining the relations [MCGR76]. Thus, we decided to enrich such bit operations as bit set/reset, bit test, and the left- or rightmost I/O bit search.

Field handling: Current high level languages provide flexible data structures. The PL/I language contains the concepts of fixed and varying size strings and such string processing operations as concatenation, extraction of a specified substring from a string, and search for a string from a specified substring [MYER78]. PASCAL is also famous for its rich data structures, such as the record consisting of unidentical fields, set, and pointer. Elements of the structures may be referred to flexibly. However, it is difficult to find any corresponding structures in the von Neumann architecture. Thus, we decided to enrich such basic field handling operations as division and concatenation of data fields.

Variable length word access: In an architecture with fixed size storage words, deciding on the word size is probably the most difficult trade-off facing the architect. If the word size is too small, the maximum value of the numbers that can be represented is too small, and larger addresses are needed. On

the other hand, larger words tend to waste storage because high order bits or digits are likely to be unused. Tagged architecture is an attractive approach to this problem [MYER78]. However, it increases the amount of storage needed, as well as the time for interpreting the tag. Variable length word accessing is a compromise between fixed size word access and tagged, fully variable size word access. It allows the user to change dynamically the word size of the main memory from one access to the next or from one application to another by using interleaved memory banks with the hardware address modifier. Thus, the word sizes of our computer are 1, 2, 4, and 8 bytes.

Table access: A two-dimensional table is a basic data structure for a variety of applications. A compiler consists of many tables. An operating system keeps many tables for the management of processes and resources. Thus, fast access to necessary data in a table is very important. We defined two primitives, namely, rowwise and columnwise accesses. These allow the user to access necessary data in a row or in a column at the same cost.



3

Basic Organization

3.1 Architectural Overview

As MUNAP employs a two-level control structure, the architecture is divided into micro- and nanolevels [BABA82b]. Figure 3.1 shows the microlevel data flow with major components. The control functions are distributed among the microprogram memory (MPM) and nanoprogram memories (NPM) in four identically structured processor units (PU). The data in processor units are transferred to/from the main memory (MM) through the interconnection network, called the shuffle exchange network (SEN). Thus, typical transfers are PU-PU, and PU-MM. The operational units included in each PU are the arithmetic and logic unit (ALU), the divide and concatenate unit (DCU), and the bit operation unit (BOU). The latter two units provide tools for nonnumeric operations. The data unit in a processor unit is 16 bits. This implies 64

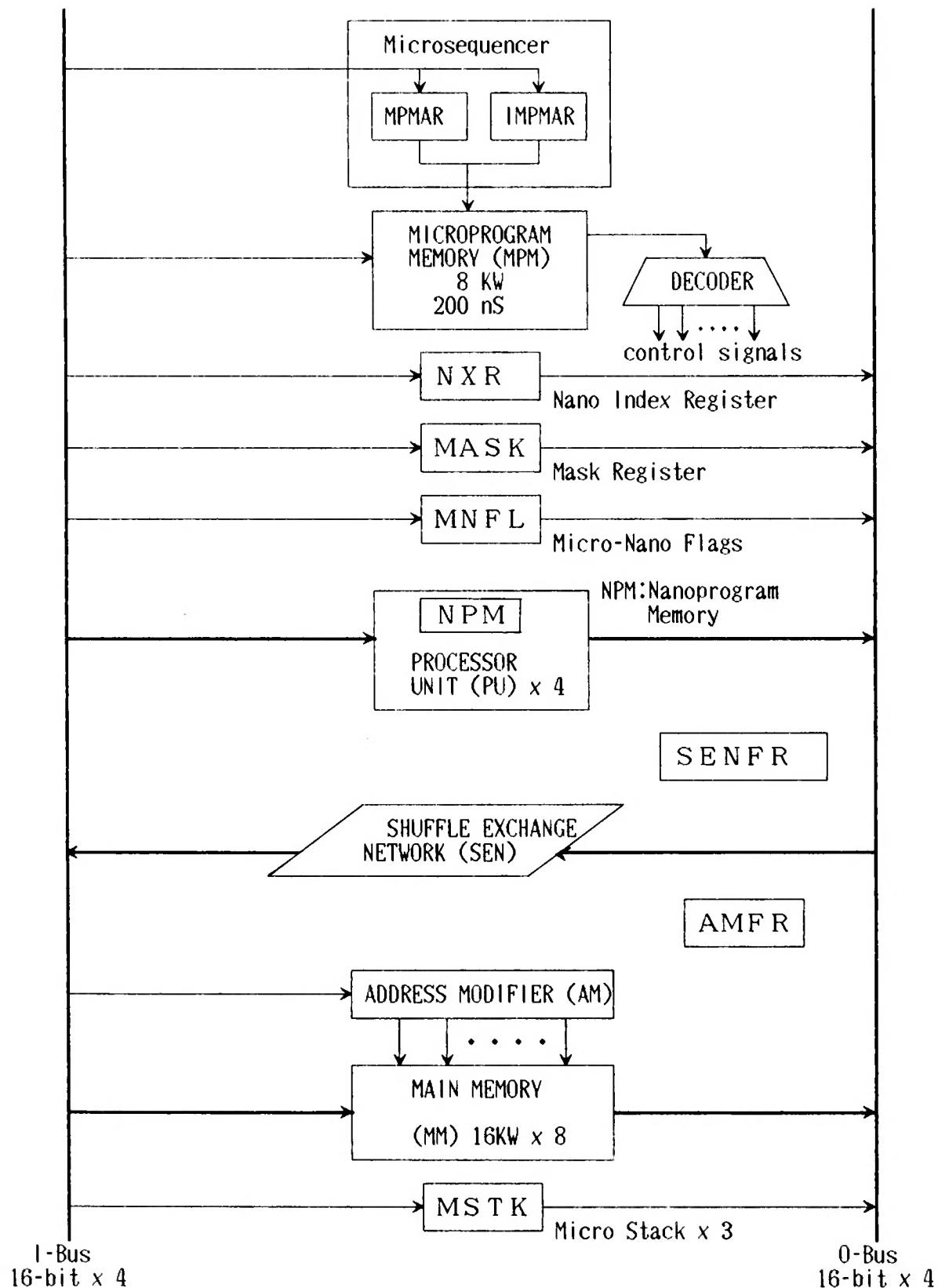


Figure 3.1
Microlevel data flow of MUNAP.

(16 x 4) bit data transfer between the processor units and the main memory through the shuffle exchange network.

Throughout the following discussions on the basic architecture of the MUNAP, the author would like to expect the readers to understand how we implement the basic scheme of a two-level microprogrammed multiprocessor control according to the previously mentioned design principles.

3.2 Microlevel Architecture

3.2.1 Hardware Organization

Microprogram Memory (MPM): The MPM is 8K words with 28-bit word length. The MPMAR and IMPMAR are MPM address registers to be used in normal run and interrupt states, respectively. The microsequencer generates a microinstruction address on them according to the microinstruction format and the state of MUNAP. The address specifies a microinstruction to be executed next. It is decoded not only to issue control signals for the other microlevel components but also to specify nanoprogram start address and the processor units to be activated.

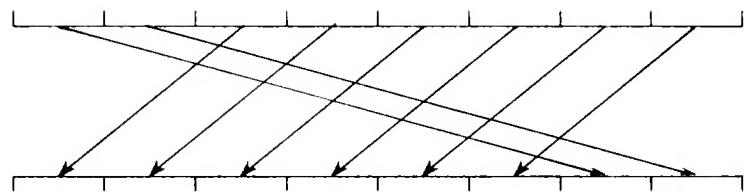
Processor Units (PU): The four identically structured processor units are viewed as parallel processing operational units at the microlevel. Each PU receives 16-bit data from I-BUS and sends the result of operations to the O-BUS. It satisfies the necessary conditions for a small computer, including a control memory, the so-called nanoprogram memory (NPM), with the sequencer, an ALU, a register file, a scratchpad memory, a

counter, and flags.

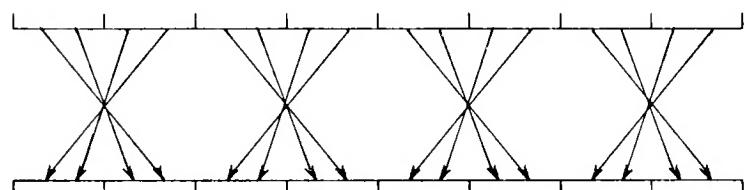
Shuffle Exchange Network (SEN): The network function is the key to the flexibility and efficiency of multiprocessor system, because it connects two major components -- processor units and main memory. The network functions include

- o cyclic shifts by 0, 4, 8, ..., 60 bits,
- o mirror transformations by 8, 16, 32, and 64 bits,
- o exchange of 16-bit data, and
- o broadcasting of 16-bit data.

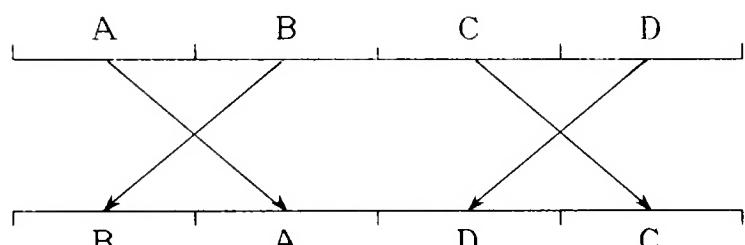
Figure 3.2 shows sample transfers from the O-BUS to the I-BUS for each of the functions. Notice that the functions are supposed not to be dedicated to special applications but to be utilized as data translation primitives for a wide range of applications. To implement these functions, the network consists of four levels of 16-number, 4-bit segment shuffle networks with exchange and broadcast cells. **Figure 3.3** shows a simplified illustration for a three-level, 8-number network. As a cell performs one of four functions, i.e., broadcast the upper or lower input to the outputs, exchange the inputs, and pass the inputs, it requires two control bits. Thus, the 32 cells are controlled by a 64-bit control word stored in the PROM, the address of which is specified by the microinstruction fields. The figure also shows an example transfer from ABCD to BABA.



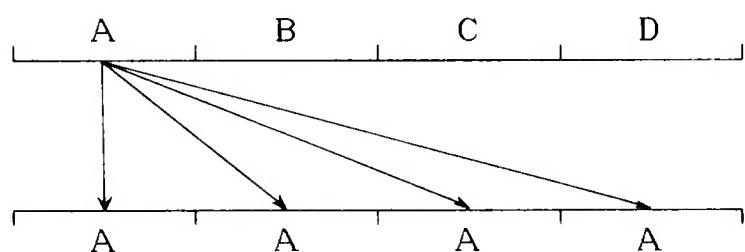
(a)



(b)



(c)



(d)

Figure 3.2 Typical network operations: (a) shift (48-bit, right); (b) mirror transformation (16-bit); (c) data exchange (BADC); (d) broadcast (AAAA).

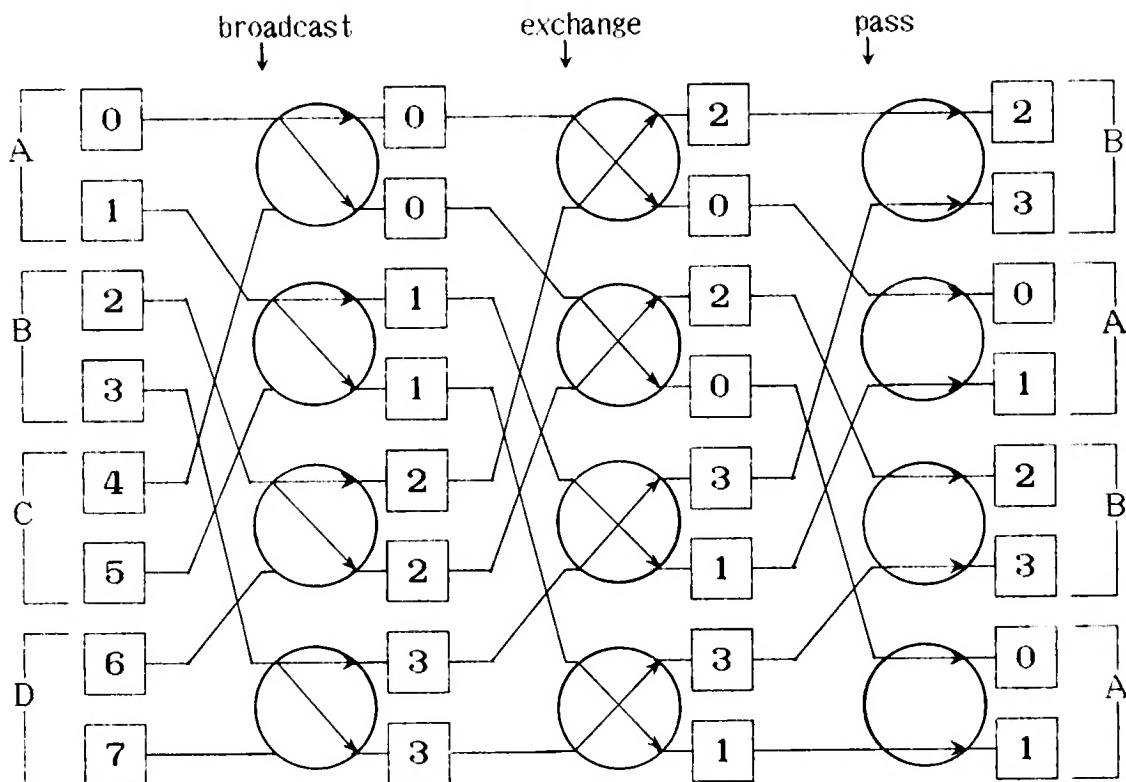


Figure 3.3
A simplified illustration of a shuffle exchange network
(3-level, 8-number).

Main Memory (MM) and Address Modifier (AM): The MM is divided into eight banks of an 8-bit wide memory module with an address register (MAR). The AM receives the logical address source from the I-BUS and produces eight physical addresses for eight MARs. Utilizing the concurrency of the access from the multiple memory banks coupled with the AM and SEN functions, various addressing facilities are provided as follows:

- o direct address from 64-bit I-BUS data (direct mode),
- o rowwise or columnwise access of 8-bit data (skewed mode), and
- o interleave access of 8, 16, 32, and 64-bit words (normal

mode).

In the *direct mode* four independent 16-bit addresses are transferred from the I-BUS to four pairs of MARs. Using four independent addresses produced at four processor units, the user can define his own addressing scheme for parallel access of four 16-bit data. An example of columnwise access in the *skewed mode* is shown in figure 3.4. If the third column is to be read, the

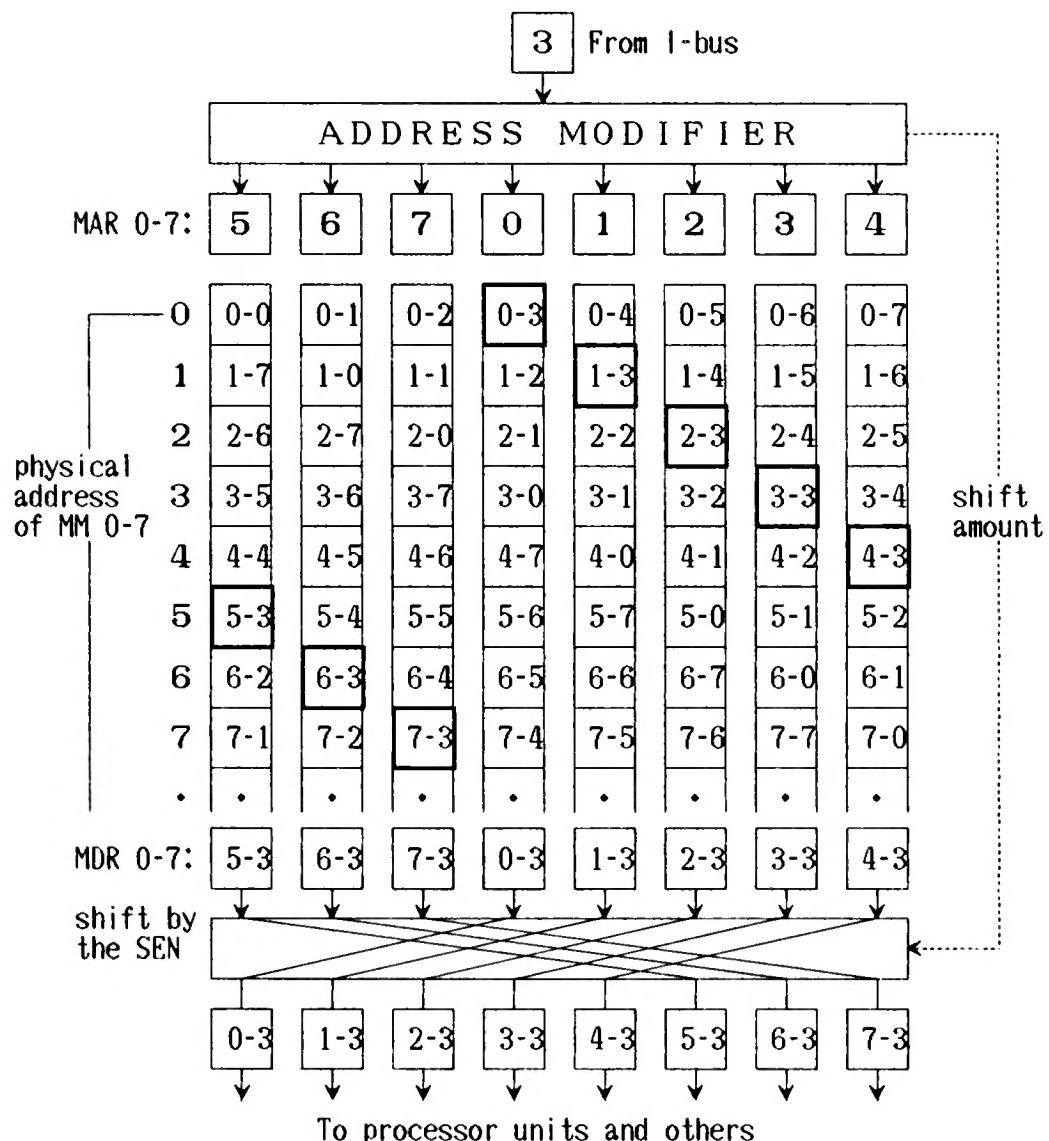


Figure 3.4
Main memory access in skewed mode.

data from 0-3 to 7-3 in the frames should be read. If the n th column is to be read, the least significant three bits, p , of the physical address for $MM\#m$ are given as follows: if $m >= n$ then $p := m-n$ else $p := m-n+8$. Practically, this computation was implemented by the simple logic operation for each memory module. Figure 3.4 also shows that the read data from the memory should be shifted for left adjustment. The shift is to be done by the SEN network, and the shift amount is given by $8 \times n$ bits. The rowwise access requires only the $(r \bmod 8) \times 8$ bits shift, where r represents a row number. The skewed mode is an elementary tool for the two-dimensional data access. In the *normal mode* the AM receives a logic address and stores addresses in the eight MARs for consecutive word access. Figure 3.5 is an example of this mode. When logic address 2 is specified, the physical addresses 1, 1, 0, ..., 0 are generated by the AM, and the data from 2 to 9 in bold faced frames are read. At the same time the shift amount of SEN is specified by the AM. Generally, the physical address p for $MM\#m$ ($m = 0 - 7$) is defined as follows: if $m >= (a \bmod 8)$ then $p := a \bmod 8$ else $p := (a \bmod 8) + 1$, where "a" is a given logic address. The shift amount is $(a \bmod 8) \times 8$. Usually, 64-bit data are written into the MM at once in these modes. However, if the user wants to write partially, the memory partial write register (MPW) enables partial writing of data into the MM banks by setting the 0's for the banks to be protected.

Stacks (MSTK) and Registers: The microstacks (MSTK 0-2) consist of 1K 16-bit words. They are used to hold the data as well as the return addresses of a microprogram. The micro-nano flags

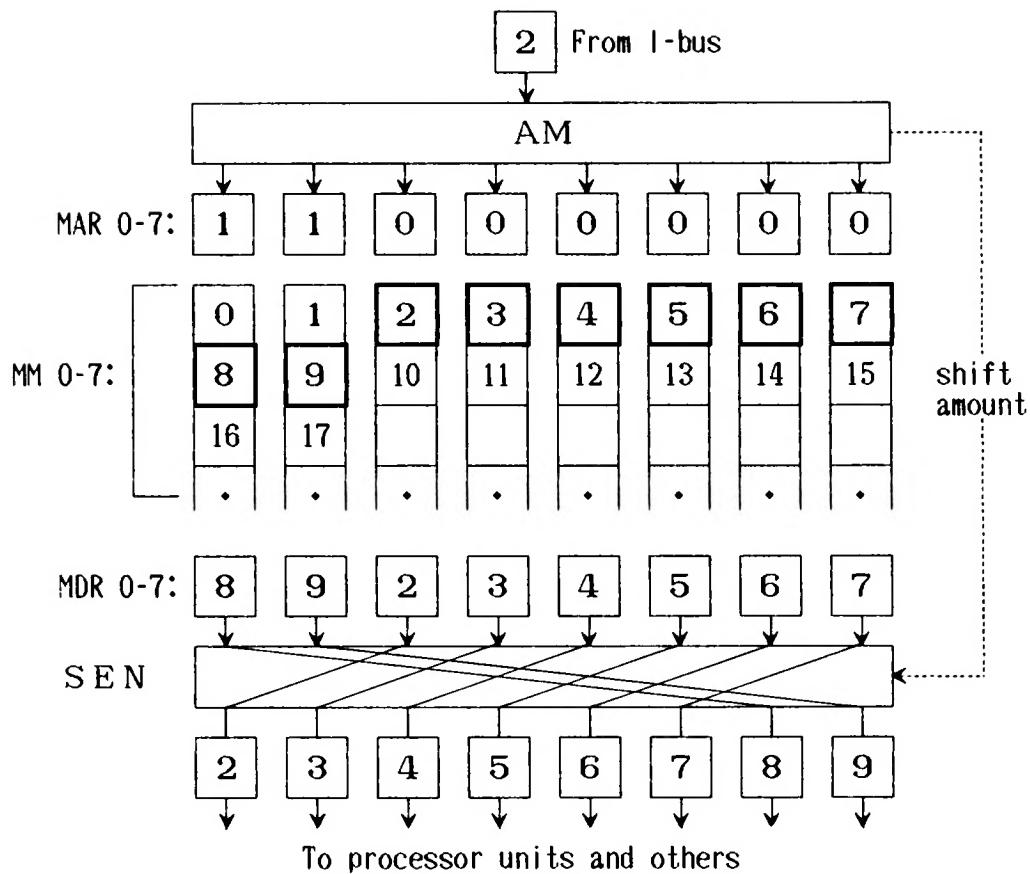


Figure 3.5
Main memory access in normal mode.

(MNFL) are used as the micro-nano interaction flags and are masked by the mask register (MASK). The nano index register (NXR) is an index register for computing the nano start address. The functional control registers, SENFR and AMFR, are provided for functionally controlling the network and the address modifier. The use of these registers will be described later.

3.2.2 Microinstruction Repertoire

In designing microinstruction formats, we decided to adopt a vertical type, where each microinstruction has a decoding control

field at the top. With this scheme the user can describe microprograms easily while utilizing the concurrency between multiple nanoprograms. The resulting microinstruction has a 28-bit width, as shown in figure 3.6.

AA, AB, BA, and CB: The AA, AB, BA, and CB microinstructions show data transfer. They may also activate the PUs by using the NXR, NA, and PFL fields, as will be described below. For example, the AA microinstruction controls such operations as reading data from the PUs onto the O-BUS, permuting them by the SEN, and writing the permuted data into the PUs. These four microinstructions differ with respect to capability of specifying the source (SA, SB, SC fields) and destination (DA, DB fields) of the transfer and the function of the SEN (SEN, SE fields).

LT: The LT microinstruction broadcasts a 16-bit literal field (LT) onto a 64-bit I-BUS by the SEN.

AM: The AM microinstruction activates nanoprograms in the PUs and sets the MAR addresses in one of the three modes, namely, direct, skewed, and normal.

BN and B: The BN and B are unconditional branch microinstructions. The BN activates nanoprograms and specifies a relative branch address to the microinstruction by the SMA field. The B cannot activate nanoprograms. Instead, it has more powerful branch functions, such as indirect branch and subroutine call/return.

TB and TBN: The TB and TBN are test branch microinstructions. The TB selects one kind of flag in the PUs and applies a logic operation to the flags selected to determine the test result. The TBN activates the specific test nanoinstructions, called NTB nanoinstructions (to be described later), and checks the flag values, which reflect the result of PU operations, to determine the succeeding microinstruction address.

MNFC: The MNFC microinstruction performs the flag set and miscellaneous controls.

MPMW: The MPMW dynamically rewrites the content of microprogram memory, thus realizing dynamic microprogramming in a narrow sense.

3.2.3 Activation of Multiple Nanoprograms

From among the 12 microinstructions in figure 3.6, 8 have 3 common fields (NXR, NA, and PFL) to specify the nanoprogram start address and activate nanoprograms as follows:

- o compute a nanostart address by adding the content of the NXR register and the NA field, and
- o send that nanoaddress to the PUs simultaneously and activate PU#*i* if bit *i* of the PFL field = 1.

The first item allows index register modification of the nanoprogram start address. Since the nanoprogram memory is reloadable, the second item allows the user to specify any combination of nanoinstructions in the PUs by specifying the same

- | | | | | | | | | |
|---|--------|-----|-----|----|-----|----|-----|-----|
| (1) AA: Nanoactivation and data transfer | 00 | SEN | SA | DA | | NA | PFL | |
| | | | | | NXR | | | |
| (2) AB: Nanoactivation and data transfer | 010 | SE | DB | SA | | NA | PFL | |
| (3) BA: Nanoactivation and data transfer | 011 | SE | SB | DA | | NA | PFL | |
| (4) CB: Nanoactivation and data transfer | 1010 | SC | DB | | | NA | PFL | |
| (5) LT: Literal generation and broadcasting | 11011 | | D | | | LT | | |
| (6) AM: Nanoactivation and MM address set | 1011 | SE | AM | | | NA | PFL | |
| (7) BN: Nanoactivation and branch | 1100 | SMA | | | | NA | PFL | |
| (8) B: Branch | 11010 | SE | P | | MAD | | REG | |
| (9) TB: Test branch | 100 | TSP | TFL | | MAD | | TPM | |
| (10) TBN: Nanoactivation and test branch | 11101 | TSP | TMA | | | NA | PFL | |
| (11) MNFC: Flags and miscellaneous control | 111100 | | | C | PMD | SR | FLG | PMN |
| (12) MPMW: Nanoactivation and MPM write | 11100 | SH | SA | | | NA | PFL | |

Figure 3.6 MUNAP microinstruction formats.

AM(5) : select an address modification mode and an address source.
 C(1) : clear the MICRO STACK #0 stack pointer.
 D(5) : literal destination.
 DA(2), DB(4) : I-bus destination.
 LT(16) : 16-bit literal to be generated on O-bus 3.
 MAD(16) : next microinstruction address.
 NA(12) : nanoprogram start address.
 NXR(1) : specify the nanoaddress index register.
 P(1) : current microinstruction address + 1 → MICRO STACK #0.
 PFL(4) : specify the processor units to be activated.
 PMD(2) : set ALL JOIN mode or PARTIAL JOIN mode.
 REG(4) : specify the functional branch register.
 SA(2), SB(2), SC(3) : O-bus source, such as processor unit, main memory, etc.
 SE(2) : a subset of the SEN field, such as 0, 16, 32, 48-bit shift.
 SEN(5) : shuffle exchange network control.
 SH(4) : 0, 4, 8, 12, ..., 60-bit shift by the SEN.
 SMA(7), TMA(4) : displacement to be added with current microinstruction address for producing next microinstruction address.
 SR(2), FLG(3), PMN(4) : set or reset (specified by SR) one kind of flags (by FLG) of specified processor units (by PMN).
 TFL(3) : select one of the flags to be tested.
 TPM(4) : specify processor units to be tested.
 TSP(2) : select the operation on the flags specified by the TFL and TPM fields.

Figure 3.6
MUNAP microinstruction formats continued (numbers in parentheses indicate the bit lengths of the fields).

nanoaddress. This results in a saving of total control storage requirements without flexibility and parallelism losses.

3.2.4 Microsequencing

Address specifications of the succeeding microinstruction and of the available conditional and unconditional branches have profound implications for system performance and cost as well as microprogram productivity. In two-level microprogrammed computers, the microlevel sequencing functions play an important role in enhancing such performance because the microinstruction can directly specify nanoprogam addresses (see figure 2.1). Therefore, the major part of sequencing functions at the nanolevel may be loaded off to the microlevel in order to avoid functional redundancy.

Thus, MUNAP provides a powerful sequencing facility at the microlevel, as summarized in table 3.1. Ordinary data transfer microinstructions, such as AA, AB, BA, and CB, simply increment MPMAR. The BN allows an unconditional branch to a microinstruction with the address of (the current microinstruction address) + (the SMA field value), while at the same time activating multinanoprograms. Notice that the actual branch occurs after the completion of activated nanoprogams. The B microinstruction allows not only an unconditional branch but an indirect branch according to the value of the PU facilities or the microlevel facilities. The branch address is computed as the sum of the MAD field and the content of the register specified by the REG field and shifted by the SEN

Table 3.1
Microinstruction sequencing functions

Type of branch	Microinstructions		Next MPMAR address
Unconditional	AA, AB, BA, CB, LT AM, MNFC		(MPMAR)+1
	BN		(MPMAR)+(SMA field)
Indirect	B		(MAD field)+(the facility specified by the REG field)
Conditional	TB	TSP field = AND/OR	if ANDed or ORed result of selected flags is true, then (MAD field); otherwise (MPMAR)+1
		TSP field = PE	if the Priority Encode is specified in TSP field, (MAD field)+(priority encoded value of MNFL flags); false action is (MPMAR)+1
	TBN	TSP field = AND/OR	if ANDed or ORed result of selected flags is true, then (MPMAR)+(TMA field); otherwise (MPMAR)+1
		TSP field = PE	true action is (MPMAR)+(TMA field)+(Priority Encoded value of TEST flag in MNFL); false action is (MPMAR)+1

according to the SE field. When the P field is 1, the next address, MPMAR+1, is pushed onto the MSTKO. An arbitrary return address may also be pushed down onto MSTK 1 and 2. These functions are utilized in forming the microsubroutines in various ways. The TB selects one of the flags by the TFL field, specifies the PUs to be tested by the TPM field, and applies one of three operations to the selected flags according to the TSP field. The TBN activates the specific nanoinstructions, called NTB nanoinstructions, by the NXR, NA, and PFL fields. The NTB nanoinstructions test the contents of resources in the PUs and set the results in the TEST flags (to be described later) of the MNFL. The results are tested by the same TBN microinstruction to determine the contents of the next MPMAR. Thus, the TBN reduces the overhead in transferring test results at nanolevel to microlevel sequencing in one machine cycle. The TB tests the MNFL flags and performs conditional sequencing. The test results of multinanoprograms may be easily merged into true or false in one step to determine the address of the succeeding microinstruction, as shown in table 3.1.

3.3 Nanolevel Architecture

Below the microlevel, there is a nanolevel processor unit architecture.

3.3.1 Processor Unit Organization

Figure 3.7 shows the organization of a processor unit.

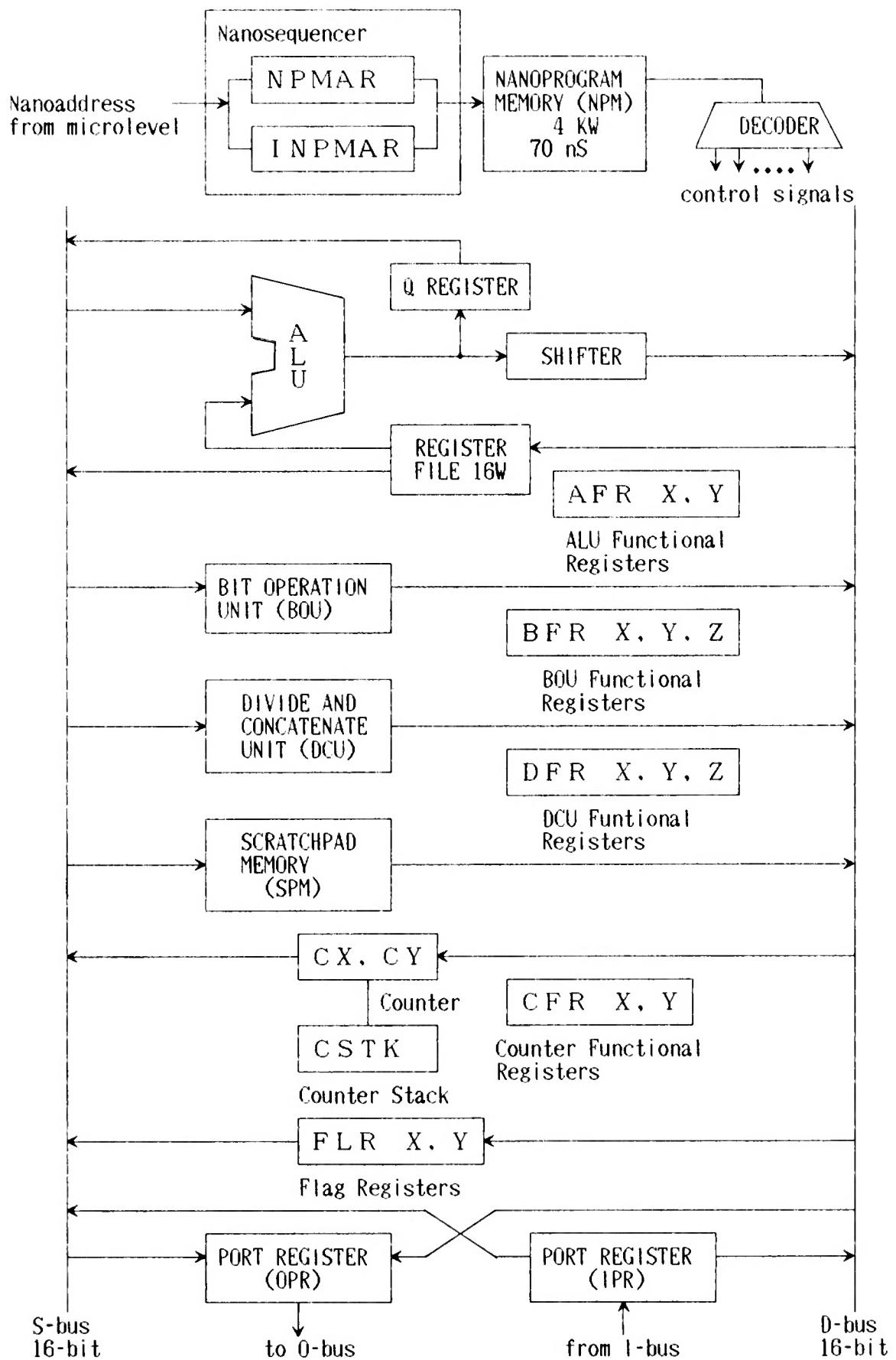


Figure 3.7
Details of the MUNAP processor units.

Nanoprogram Memory (NPM): The capacity of the NPM is 4K words, with 40-bit word length. In the same way as the microlevel, the NPMAR and INPMAR are attached as the NPM address registers to be used in normal run and interrupt states. The nanosequencer generates a nanoinstruction address in them according to the nanoinstruction format and the MUNAP state. As the major part of sequence control functions are loaded off to microlevel, the function of the nanosequencer becomes simple. The nanoinstruction is decoded to issue control signals to the other parts of the processor unit.

Arithmetic and Logic Unit (ALU): The ALU covers ordinary operations, such as add, subtract, and other logical operations. This part is composed of four 4-bit slice chips, AMD (Advanced Micro Devices, Inc.) Am2903 [ADVA79], which include such associated units as shifter, Q register, and 16-word register file. The register file (RF) may be used for temporary storage. The ALU receives two inputs (one from the S-BUS and the other from the RF), performs an operation, and sends the result to the shifter or the Q register. The shifter output is sent to the D-BUS. The Q register is intended to be used for special purpose Am2903 functions, such as multiplication and division.

Bit Operation Unit (BOU): The purpose of the BOU is to enable bit operations on 16-bit data as follows:

- o to set the specified bit with one or zero,
- o to test the specified bit, and
- o to indicate the bit position of the most or least significant

one or zero.

The bit search of the last function may be started from either the left end or the right end of given data. It resembles an IC (Integrated Circuit) chip function of the so-called priority encoder. Each function may be completed in one machine cycle, while ordinary machines require several steps of shift and test.

Divide and Concatenate Unit (DCU): The purpose of the DCU is to provide facilities for field handling operations. Its structure and operations are shown in **figure 3.8**. Receiving 16-bit data from S-BUS, it

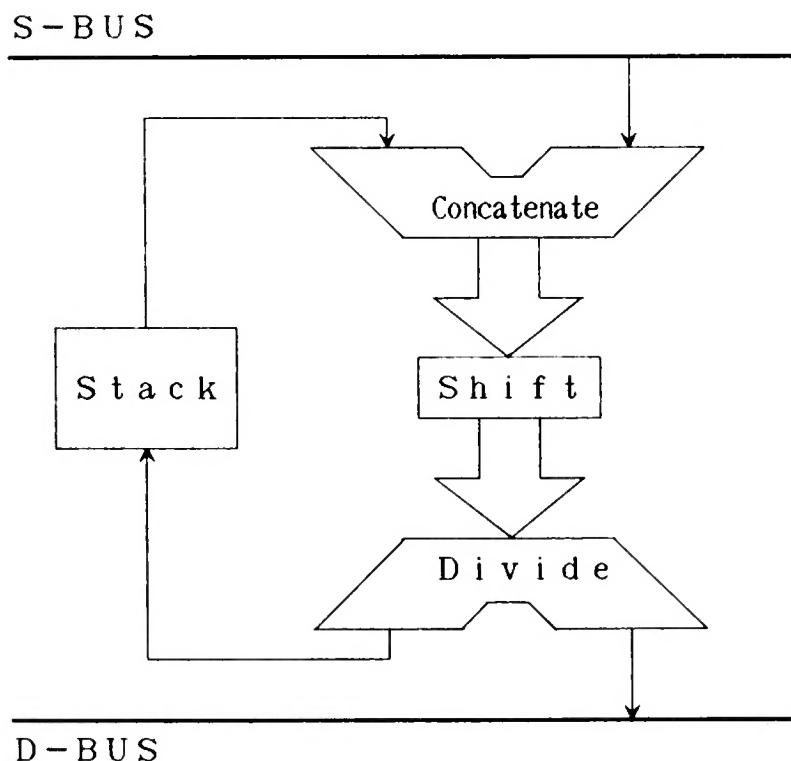


Figure 3.8
Divide and concatenate unit.

- o shifts and divides the input data into two parts at the specified bit position;
- o shifts and divides the input data and pushes one onto the DCU stack;
- o concatenates a part of the input with zeros at the specified bit position and shifts it; and
- o concatenates a part of the input with the top of the DCU stack and shifts it.

The four DCUs in the PUs, combined with the SEN, perform nonnumeric operations, such as extracting and embedding the data field, which in conventional machines are done by tedious masking and shifting operations. The DCU stack may be used to store some data until processing of the rest of the field is completed, and it rapidly generates the result by concatenation.

Scratchpad Memory (SPM): The capacity of the SPM is 16-bit 1K words. The first 256 words may be addressed directly by a nanoinstruction field or indirectly by 8-bit counters CX or CY. The whole part may be accessed by specifying the address by 16-bit counter C, which is the concatenation of CX and CY. This facility is to be used as the largest temporary storage in a PU. Using the C counter as a pointer, it may also be used as a stack. The SPM stack has been utilized in many of the applications to be described later.

Counters (CX, CY, and C): The CX and CY are 8-bit counters. The CY has a 16-word stack to be used for saving counter values. Because used with a hierarchical microsubroutine structure, the

stack enables the user to save a value at each level by just one microorder. The counters may be used as a concatenated 16-bit counter, called C.

Flag Register (FLR): The FLR is a 32-bit flag register and is divided into two 16-bit parts, called FLRX and FLRY. The bits basically indicate the status of facilities in the PU. For example, the flags for the ALU are ALU carry, ALU sign, ALU overflow, etc. Two general purpose flags are also included in the registers.

Functional Control Registers (AFR X & Y, BFR X, Y, & Z, DFR X, Y, & Z, and CFR X, & Y): According to the design principle of uniformity, we have attached functional control registers for the major units of ALU, BOU, and DCU. The registers from AFRX to DFRZ correspond to nanoinstruction fields, and, when a functional control is specified, the value of a register is used as control bits instead of the corresponding nanoinstruction field. The contents of CFRX and CFRY are compared with those of the counters CX and CY to check the end of count dynamically.

Port Registers (IPR and OPR): These registers connect the nanolevel data flow with the microlevel one. The IPR receives 16-bit data from microlevel I-BUS 0, 1, 2, or 3 and sends it to a nanolevel bus of S-BUS or D-BUS. The OPR receives data from S-BUS or D-BUS and sends it to microlevel O-BUS 0, 1, 2, or 3. The I-BUS#i and O-BUS#i are connected to the port registers of PU#i.

3.3.2 Nanoinstruction Repertoire

We designed 40-bit vertical nanoinstructions, as shown in figure 3.9. They have one bit field in common, called ENP (End of NanoProgram), which indicates the end of nanoprogram. Thus, a nanoprogram may end at any nanoinstruction to return the control to microlevel.

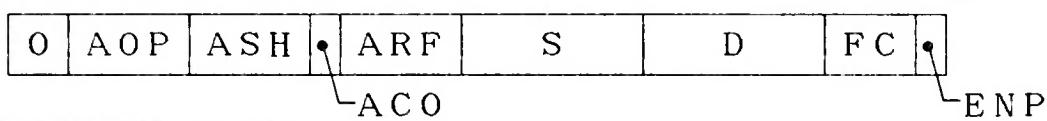
ALU, BOU, and DCU: The basic organizations of the nanoinstructions are the same. They get input data from a facility (specified by the S field) through S-BUS, perform an operation on it, and store the result in a facility (specified by the D field) through D-BUS. The operation is controlled by the leftmost fields of the nanoinstructions. The FC field controls the flags and counters in parallel with the above microoperations.

EX: The EX covers external data transfers between microlevel and nanolevel, and exceptional operations, such as special purpose flag set. It contains many unused operation codes for the future extension.

NLT: The NLT generates a 16-bit literal and sends it to a facility specified by the D field.

NTB: The NTB nanoinstruction selects two flags (using the TS1 and TS2 fields) from the 32-bit flags in the FLR register. Then the logical operation (TSC field) is performed on the flags to determine the test result.

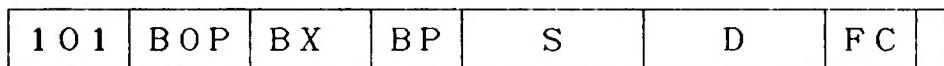
(1) ALU: ALU operation



(2) DCU: DCU operation



(3) BOU: BOU operation



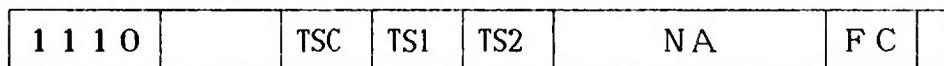
(4) EX: External operation and extension of internal operation



(5) NLT: 16-bit literal



(6) NTB: Conditional or unconditional branch



(7) NOP: No operation



ACO(1) : enable carry out into the upper processor unit.

AOP(5) : provide ALU function control.

ARF(4) : address of the register file.

ASH(5) : specify ALU shifter function when AOP \neq 0;
specify ALU functions for multiplication and division when AOP=0.

BOP(3) : BOU operation.

BP(5) : bit position.

BX(5) : cooperate with the BOP field by specifying the value (0/1) to be set,
counted, tested or priority encoded.

D(9) : D-bus destination.

DOP(3) : DCU operation.

DP(5) : bit position.

DSH(5) : shift amount.

ENP(1) : indicate the end of nanoprogram.

EX(12) : external operation or internal operation to be extended.

FC(5) : miscellaneous control. increment or decrement counters; control the
counter stack; disable the flag set; set or reset general purpose
flags in the flag register.

NA(12) : nano branch address.

NLT(16) : 16-bit literal to be generated on S-bus.

S(9) : S-bus source.

TS1(5), TS2(5) : select one of 32 flags in the flag register.

TSC(3) : logical operation, such as AND, OR, EXCLUSIVE OR, on the two flags
specified by the TS1 and TS2 fields.

Figure 3.9

Nanoinstruction formats (numbers in parentheses
indicate the lengths of the fields).

NOP: The NOP is a nooperation nanoinstruction mainly used for debugging hardware and firmware.

3.4 Interaction between Microlevel and Nanolevel

We have described the architectures at micro- and nanolevels in the preceding sections as if they were independent. However, they cooperate with each other to perform given jobs. They may be considered from the three viewpoints of control, data flow, and timing.

3.4.1 Micro-Nano Interaction Mechanism

The control flow has been implemented by means of a unified mechanism, called a micro-nano interaction mechanism, illustrated in figure 3.10. In this figure the micro-nano facilities are abstracted to emphasize the interaction between two levels. For example, the microlevel modules represent such facilities as main memory, shuffle exchange network, etc., which may be seen at microlevel data flow, shown in figure 3.1. The micro-nano flags (MNFL) are placed as an interface and play an important role by sending and receiving information between the two levels. The MNFL consists of the following six kinds of flags:

- o Nanohalt (**NHLT**): the end of nanoprogams of four PUs;
- o Microrequest (**MREQ**): the request from microlevel to nanoprogams;
- o Test (**TEST**): the result of tests by nanoprogams;
- o Nanorequest (**NREQ**): the request from nanoprogams to microprogram;

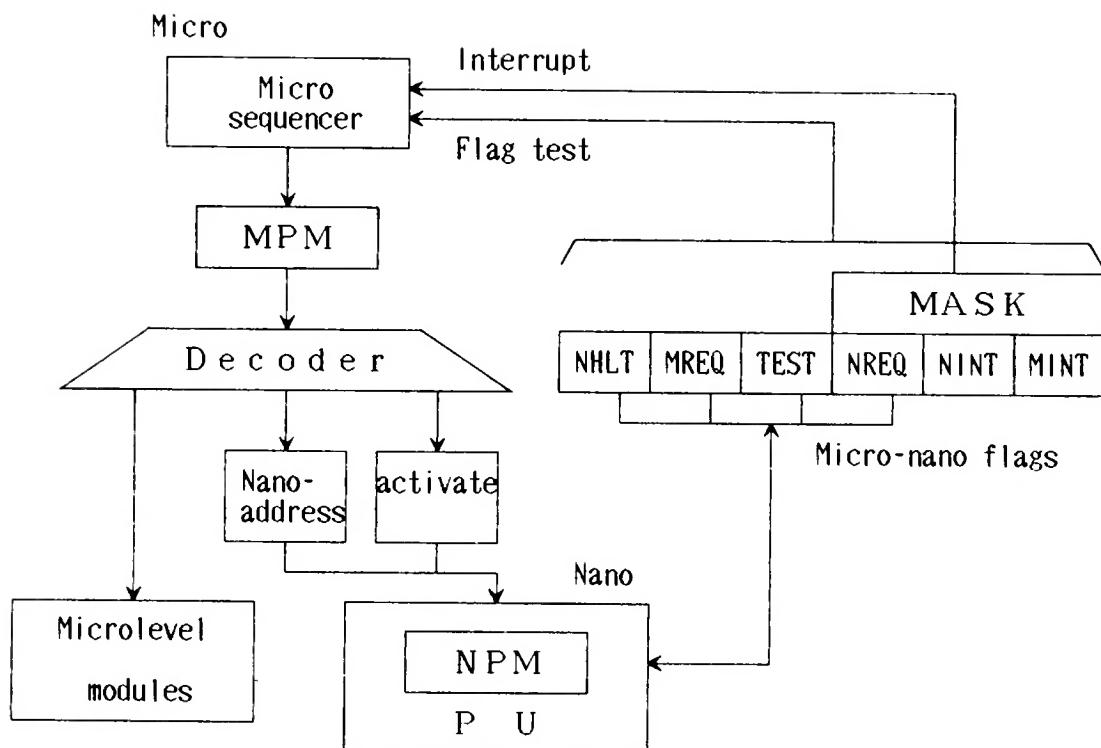


Figure 3.10
Micro-nano interaction mechanism.

- o Nanointerrupt (**NINT**): the interrupt caused by nanoprogram execution; and
- o Microinterrupt (**MINT**): the interrupt caused by microprogram execution.

Each kind of flag, except MINT, is 4-bit, corresponding to four PUs. Operations for the flags are divided into three categories. For each category, operations are described first.

Activation and Termination of Nanoprograms

Micro → Nano: Specify nanoaddress and activate parallel nanoexecution paths (by NXR, NA, PFL fields of microinstruction).

Nano → NHLT → Micro: Set the NHLT flag in MNFL to notify the end of nanoprogram (by the ENP field of nanoinstructions).

As shown in figure 3.10, the decoded microinstruction fields (NXR, NA, and PFL) specify a nanoprogram start address and the processor units to be activated. The ENP field of the last nanoinstruction of the activated nanoprogram should be 1 to set the NHLT flag and indicate the end of nanoprogram. A hardware mechanism is provided to check the NHLT flags for four PUs and start the next microinstruction execution. Figure 3.11 shows the interaction between two microinstructions (mI) and their nanoprograms, consisting of a few nanoinstructions (nI). The special mark (▲) of the last nanoinstruction indicates that the ENP field is set to 1.

Notice that the overlap of operations at the two levels yields the following three cases of microcycles:

- a. Only the micro is running.
- b. Only the nanos are running.
- c. Both the micro and nanos are running in parallel.

These cases are clearly seen in figure 3.11. Case (a) happens if microinstructions LT, B, TB, and MNFC, which do not activate nanoprograms, are running. Case (b) happens if nanoprograms with multiple nanosteps are running after being activated by a microinstruction (see machine cycles 1, 2, and 3 in figure 3.11). Case (c) represents parallel operations both at micro and the activated nanos (see machine cycle 4 in figure 3.11).

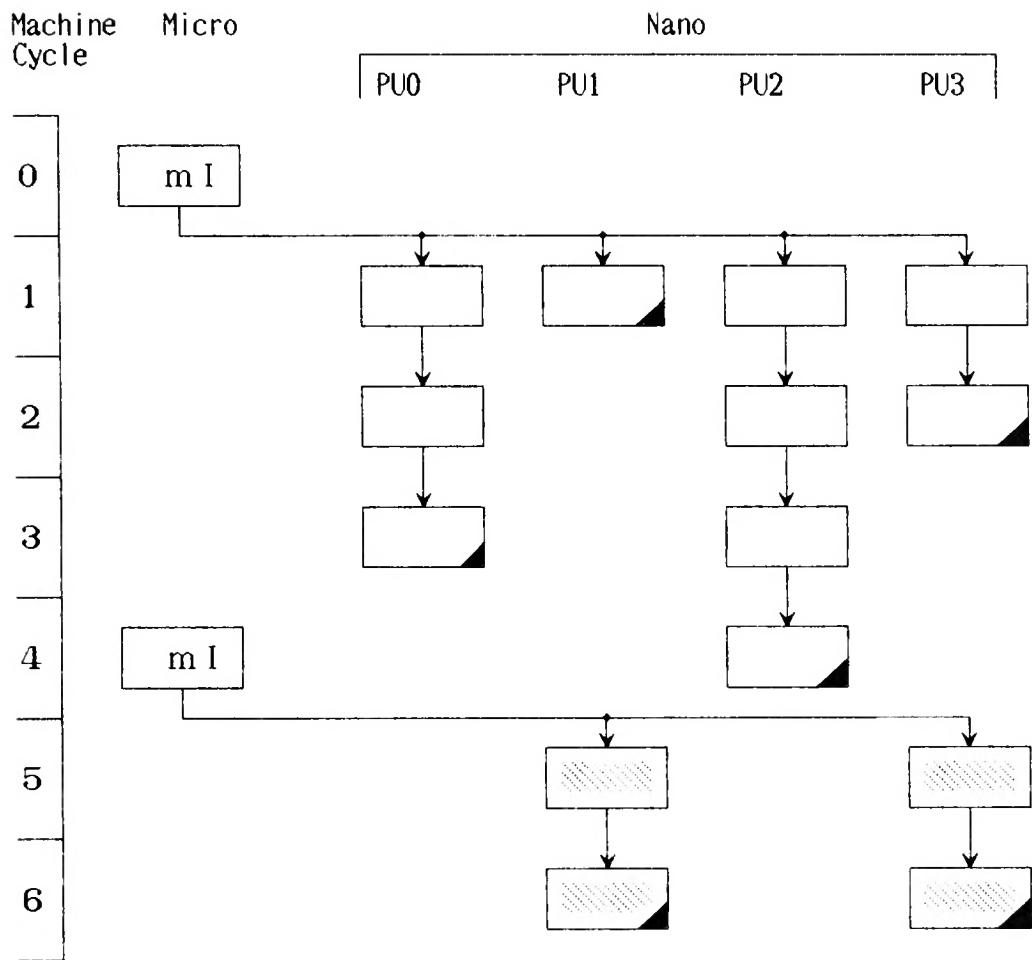


Figure 3.11
Nanoprogram activation in all join mode.

A simple condition for a microinstruction to start its execution is to wait for the end of all the nanoprograms activated by the preceding microinstruction, as shown in figure 3.11. This is detected by all the NHLT flags becoming one. However, in considering the interaction, we found that the succeeding microinstruction may start its execution when the processor units, to be activated by the specification of the PFL field of the microinstruction, complete their current nanoprogram executions. Figure 3.12 depicts the difference between the two modes, using the same micro- and nanoprograms shown in figure

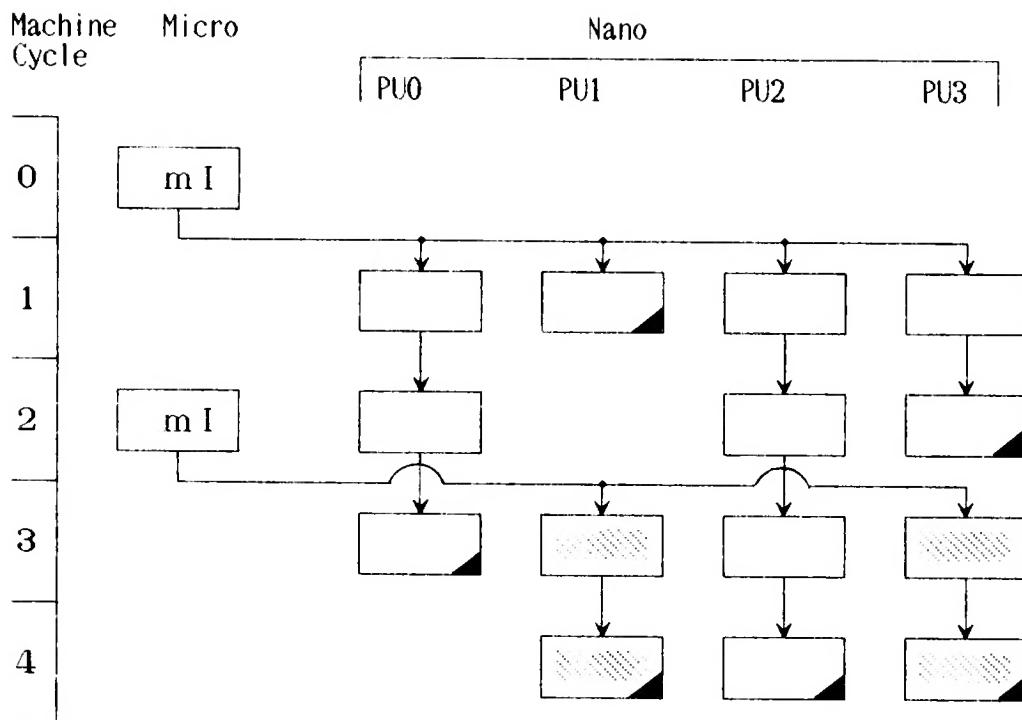


Figure 3.12
Nanoprogram activation in partial join mode.

3.11. The second microinstruction can start its execution at machine cycle 2. Under the former restricted condition, this mode is the *All Join Mode*, while under the latter, looser condition, it is the *Partial Join Mode*. The user can control these modes by setting a flag, called the PMD flag. The advantages of the partial join mode over the all join mode are as follows:

1. Needless to say, the loose start condition of the partial join mode enables an earlier start of succeeding microinstructions for the partial join mode than for the all join mode. It enhances the parallelism and makes the execution speed faster. As shown in figure 3.12, the partial join mode realizes

a speedup of 2 machine cycles.

2. In the partial join mode, a microprocess and multiple nanoprocesses can perform their executions independently, while such performance is quite restricted in the all join mode.

In a later section, the reader will see an example of the effective use of the partial join mode for parallel processing of Prolog.

Transferring the Test Results

Nano → TEST: Set the TEST flag in MNFL to notify the nanolevel test results (by the NTB nanoinstruction).

TEST → Micro: Test the TEST flag (by the TB, TBN microinstructions).

As there are two sequencers, i.e., micro- and nanolevel ones, the result of the test by the NTB nanoinstruction is used not only for sequencing a nanoprogram but also for sequencing the microprogram. In the former case, the NTB tests the flag register (FLR) in the PU and determines the next nanoinstruction address. In the latter case, the ENP field of the NTB nanoinstruction is set to 1 to send the test result to microlevel.

Request from One Level to Another

Micro → Nano: Set the MREQ flag to transmit the request from the microlevel to each processor unit.

Nano → Micro: Set the NREQ flag to state the request from the nanolevel.

Usually, micro- and nanolevels are tightly coupled to perform a job as a single computer. Their interactions are statically described as micro-nano combined operations and they do not need to send "request" signals to cooperate with each other. According to circumstances, however, it is convenient that the microprogram and four nanoprocesses define independent processes, named micro- and nanoprocesses, respectively. This is possible because the micro- and nanolevels have their own control memories and sequencers. The processes, run concurrently at microprogram level, are expected to show new applications.

The above operations are to be used for this purpose as the tool for micro-nano interprocess communications. The request sets MREQ and NREQ to interrupt each other. The IMPMAR and INPMAR registers are provided for switching the address registers according to changes in states of micro- and nanolevels (see the MPM and NPM address registers in figures 3.1 and 3.7). The other sources of the interruption are overflow of the ALU operation or stacks in both levels that set the MINT and NINT flags. The interrupts by NREQ, NINT, and MINT may be masked by the MASK register.

As to synchronization at the nanoprogram level, it should be noted that the microcontrolled facilities and the nanocontrolled ones are disjoint except for the port registers (see figures 3.1 and 3.7). This means that the microprogram can read and write

facilities in a processor unit only by activating its nanoprogram. The nanocontrolled facilities in the four PUs are also mutually disjoint, and a nanoprogram cannot directly read or write facilities in the other PUs, but must ask the microprogram to do it. Further, the hardware mechanism keeps a new nanoprocess waiting until the current nanoprocess of the same processor unit has completed its execution. These disjoint uses prevent such conflicts as a processor unit resource writing while it is being read. When the micro and each nano interact through the port registers, they must communicate explicitly by using a request mechanism because the port registers are the only resources that can be read or written by both micro and nano independently (i.e., the port registers are in a critical region between micro and each nano) [HANS73]. The hardware mechanisms, combined with firmware micro- and nanoprograms, support the synchronization.

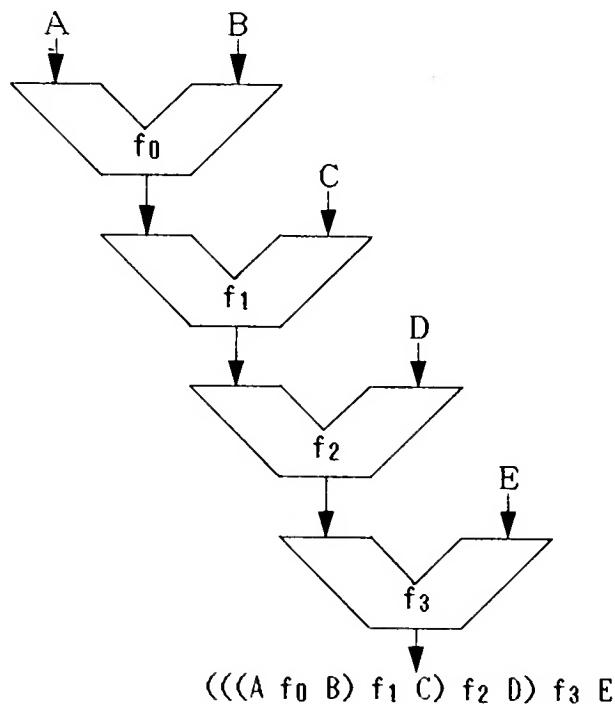
3.4.2 Micro-Nano Data Transfer Mechanism

The data transfer mechanism has important implications for the efficiency of a parallel processor system. Many network configurations have been investigated for this purpose [FENG81]. We selected a shuffle exchange scheme as the basic frame of the network to be used for connecting nanolevel processor units and such microlevel facilities as main memories.

The next concern is directed toward the compatibility of parallel and serial operations. It is well-known that even the so-called parallel problems, such as matrix computation, include

serial operations. If we design a machine to be applied to a wide range of problems, it should be flexible enough to provide operations of multiprocessor units, not only in parallel but also in series. This will reduce the overhead for applications that require serial processing. Here, the term serial means the serial execution of mutually dependent operations of four PUs on the source data (for example, extract a data field from a block in a PU and count ones in the field in another PU). Without such an operation, the sequential processes would be performed in only one processor unit, resulting in lower processor unit utilization.

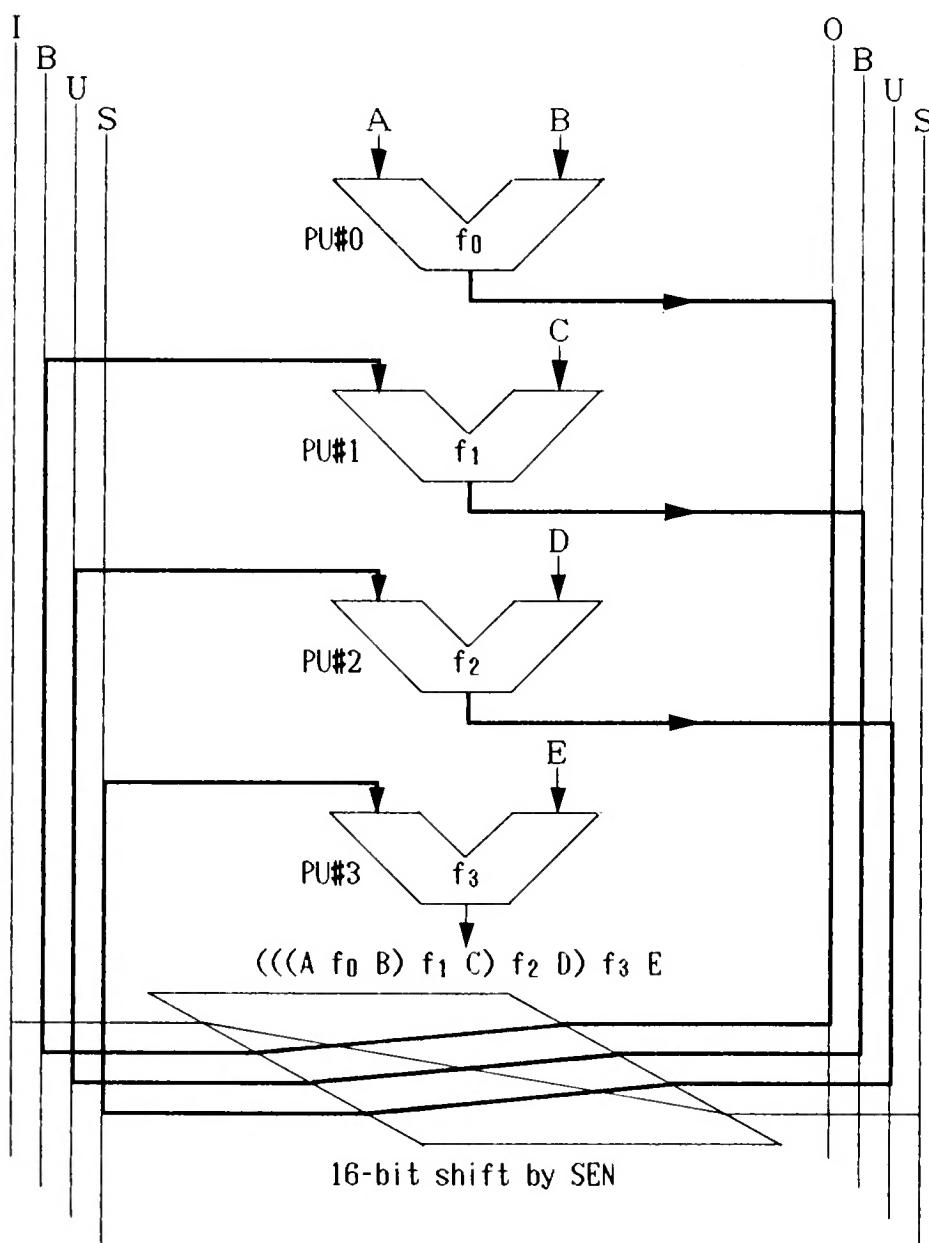
Figure 3.13a shows an example of four-level, 16-bit serial operation. Each f_i ($i = 0, 1, 2, 3$) may be the ALU, DCU, or BOU



3.13a

Figure 3.13
Four-stage 16-bit serial operations and its data flow:
 (a) four-stage 16-bit serial operation; (b) data flow for (a).

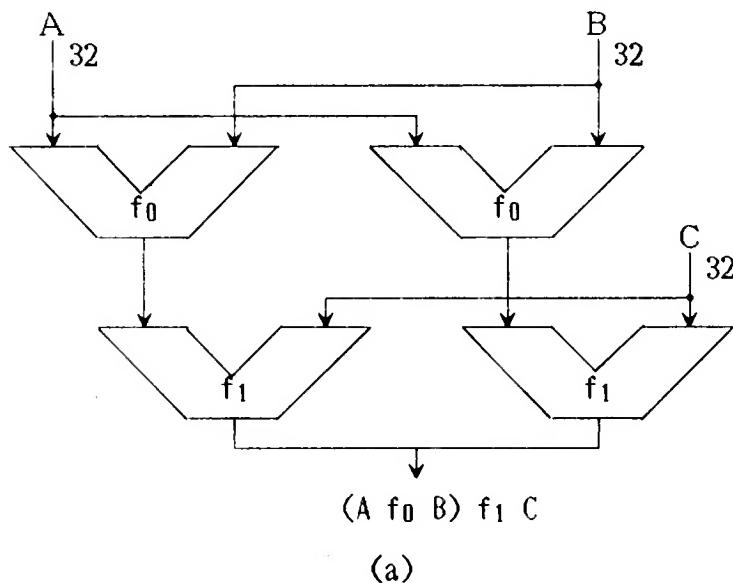
operation of each processor unit. Figure 3.13b illustrates the physical data flow for the operation shown in figure 3.13a. The SEN network function is a 16-bit cyclic shift. The operation, i.e. additions, are controlled independently by each nanoinstruction. The data are circulated according to the



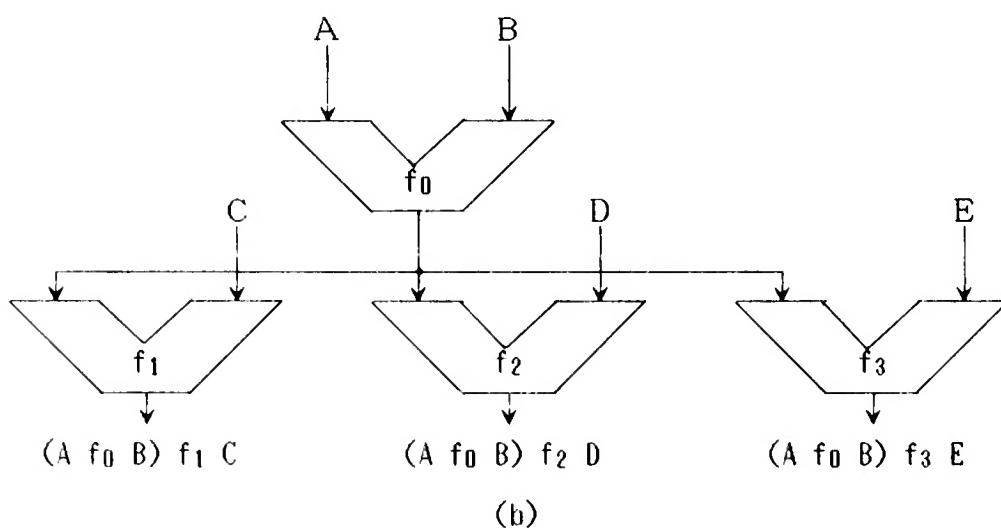
3.13b

arrows, and the circulation is controlled by the microinstruction.

Selection from various network functions also provides such useful operations as two levels of 32-bit operation (figure 3.14a) and two levels of a 16-bit broadcasting (figure 3.14b).



(a)



(b)

Figure 3.14
Various serial operations: (a) two-stage 32-bit operation;
(b) two-stage 16-bit broadcast.

These may be extended further by adding new control patterns to the PROM control memory of the SEN.

3.4.3 Timing Analysis of Two Levels

The basic idea for the reduction of the overhead of two-level control is to overlap the two levels of operations for control words as shown in figure 3.15. Each micro-nano instruction pair requires ten clock intervals for its completion, and the first four clocks are overlapped with the execution of the previous microinstruction. Therefore, a machine cycle consists of six phases, phases 1-6, and in the phases the following operations

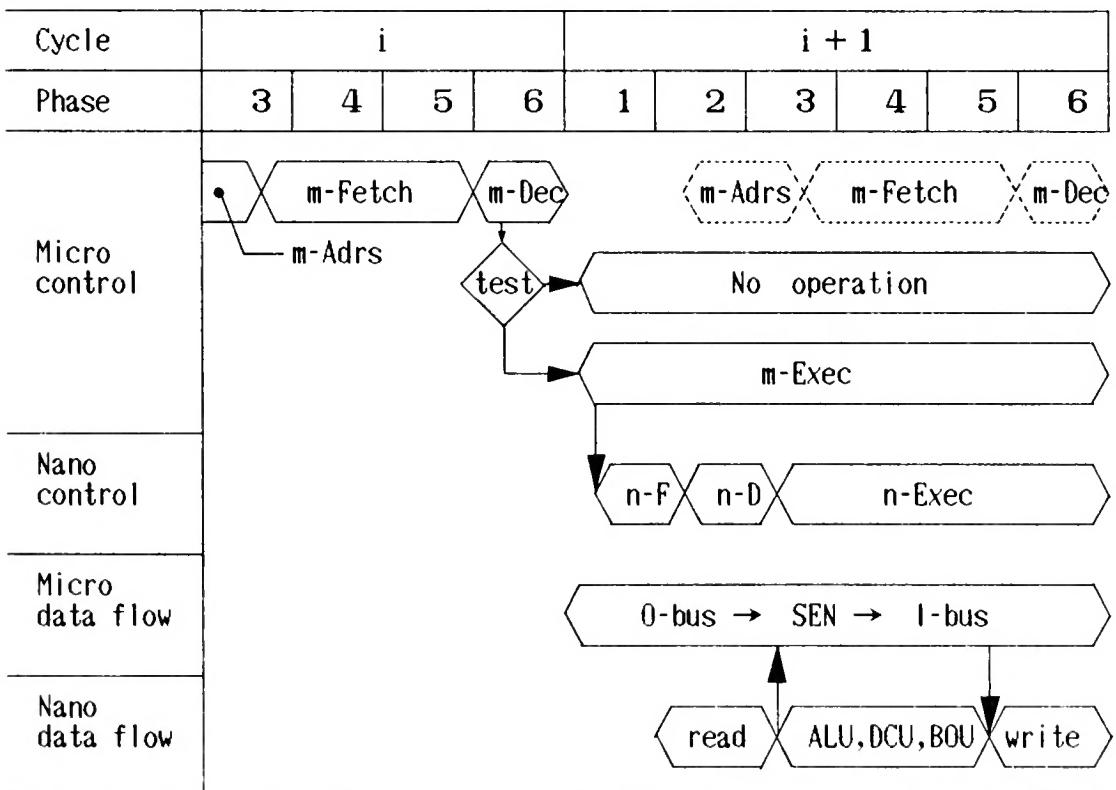


Figure 3.15
Timing analysis of a data transfer microinstruction.

are done (the expressions in parentheses correspond to those in figure 3.15):

- o (Phases 3-5)_{ad}: A microinstruction is read out (m-Fetch).
- o (Phase 6)_{ad}: The microinstruction is decoded (m-Dec); the NHLT flags in MNFL and the PMD flag are tested to determine whether a new micro- or nanoprocess is to be started (test).
- o Phase 1: The decoded signals are issued from microlevel and a nanoinstruction is fetched (n-F).
- o Phase 2: The nanoinstruction is decoded (n-D) and nanolevel units are read out; the nano test results are transferred to microlevel as described above and are used for determining the successive microinstruction address (m-Adrs).
- o Phases 3-5: Nanolevel functional units operations (n-Exec) and microlevel network control (m-Exec).
- o Phase 6: The results of functional operations by ALU, DCU, or BOU are written into nanolevel units or microlevel units (write).

Each phase of the current MUNAP requires about 90 nanoseconds. Therefore, one microinstruction and four nanoinstructions may be concurrently executed in a minimum of 550 nanoseconds. This relatively low speed is due mainly to the use of LS TTL ICs. Some microinstructions, such as TBN and a microinstruction that controls the above mentioned serial operations, require more time than the basic machine cycle. Such cases are automatically detected by hardware in order to stretch a specific phase; thus, the user does not need to know the case.

3.5 Microprogram Examples*

We have described the two levels of architectures and their interaction mechanism. The reader might be inclined to conclude from this description that the machine is very complex. We shall offset this conclusion by presenting several example microprograms. Each example includes the contents of the operations, the microprogram in mnemonic form, and the data flow. Notice that ENP stands for the end of a nanoprogram.

Example 1: one micro activates a nanoprogram The microinstruction at address $(100)_{16}$ activates a nanoprogram at address $(20)_{16}$ in processor unit PUO. The nanoprogram consists of two nanoinstructions, nI(20) and nI(21), where 20 and 21 are the nanoaddresses in hexadecimal format. The first nanoinstruction adds the contents of RFO(1) and SPM0(11), and the second performs 'logical AND' of RFO(5) and SPM0(10). RF_i(j) and SPM_i(j) stand for register file and scratchpad memory at address j in PU_i, respectively.

nI(100): activates a nanoprogram at address (20) in PUO
 [AA: SEN, SA, DA are not used, NXR=0, NA=20, PFL=8]

nIO(20): SPM0(10) <- RFO(1) <+> SPM0(11) go to next
 [ALU: AOP=ADD, ARF=1, S=SPM11, D=SPM10, ENP=0]

nIO(21): RFO(2) <- RFO(5) <AND> SPM0(10) ENP.
 [ALU: AOP=AND, ARF=5, S=SPM10, D=RF2, ENP=1]

*This section can be skipped for the first reading.

The NA field of the microinstruction specifies the nano start address. The PFL equals 8 (1000 in binary), which activates the corresponding processor, PUO.

Example 2: an SIMD operation The microinstruction at address (101) activates the same four nanoprograms at address (22) in four processor units. They perform the same operation as in example 1.

mI(100): activates four same nanoprograms at address (20) in PUO [AA: SEN, SA, DA are not used, NXR=0, NA=20, PFL=F]

```

nI0(22): SPM0(10) <- RFO(1) <+> SPM0(11) go to next
          [ALU]

nI0(23): RFO(2) <- RFO(5) <AND> SPM0(10), ENP.
          [ALU]

nI1(22): SPM1(10) <- RF1(1) <+> SPM1(11) go to next

nI1(23): RF1(2) <- RF1(5) <AND> SPM1(10), ENP.

nI2(22): SPM2(10) <- RF2(1) <+> SPM2(11) go to next

nI2(23): RF2(2) <- RF2(5) <AND> SPM2(10), ENP.

nI3(22): SPM3(10) <- RF3(1) <+> SPM3(11) go to next

nI3(23): RF3(2) <- RF3(5) <AND> SPM3(10), ENP.

```

The mnemonics for the nanoinstructions are omitted because they are the same as those in example 1. Notice that the PFL field of the microinstruction is changed to F (=1111 in binary) to activate all four nanoprograms. This is a typical example of SIMD type operations where the multiple processors perform the

same operation on multiple data.

Example 3: an MIMD operation The microinstruction at address (102) activates four nanoprograms, which do different operations; thus, this is an example of an MIMD operation.

mi(102): activates four different nanoprograms at address (24)
[AA]

nI0(24): SPM0(10) <- RFO(1) <+> SPM0(11), ENP.
[ALU]

nI1(24): SPM1(10) <- (2345)₁₆, ENP.
[LT]

nI2(24): RF2(3) <- divide the data in SPM2(30)
[DCU]

SPM2(10) <- bit count in RF2(3), ENP.
[BOU]

nI3(24): RF3(2) <- RF3(2) + SPM3(12), CY3 - 1
[ALU]

(25): IF CY3 <> 0 THEN GO TO nI3(24)
[NTB]

(26): SPM3(12) <- RF3(2), ENP.
[ALU]

Example 4: parallel data transfer and serial operation The microinstruction (103) activates four PUs to transfer the data in PU2,3 to PU0,1. Figure 3.16 shows the simplified data flow between PU0,1 and PU2,3, controlled by the mi(103). Further, the microinstruction (104) activates four PUs to perform a serial operation. The data flow for the serial operation by the mi(104) was shown in figure 3.13b.

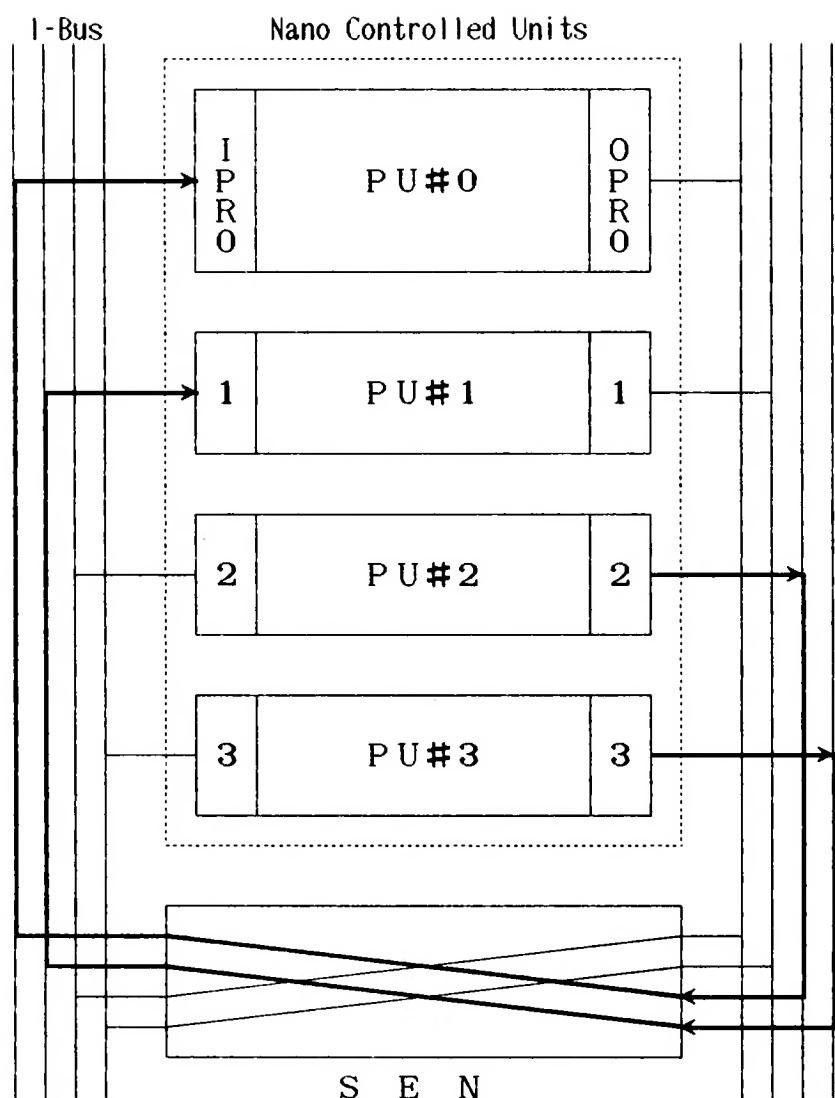


Figure 3.16
Parallel data transfer.

mI(103): 32 bit shift by the SEN network; activates nanoprocessors at address (27)
 [AA]

nIO(26): SPM0(1) <- IP0, ENP.
 [EX]

nI1(26): SPM1(1) <- IP1, ENP.
 [EX]

nI2(26): OPR2 <- SPM2(1), ENP.
 [EX]

nI3(26): OPR3 <- SPM3(1), ENP.
 [EX]

mI(104): 16 bit shift by the SEN network; activates nanoprocessors at address (27).

nIO(27): OPRO <- RFO(1), ENP.
 [ALU]

nI1(27): OPR1 <- RF1(1) <+> IP1, ENP.
 [ALU]

nI2(27): OPR2 <- RF2(1) <+> IP2, ENP.
 [ALU]

nI3(27): SPM3(1) <- RF3(1) <+> IP3, ENP.
 [ALU]

Example 5: main memory read and write The AM microinstruction (105) sets MARs and (106) reads MM. The data read from MM are transferred to SPMs through SEN and IPR. The microinstruction (107) sets MARs and (108) writes data into MM. The data are transferred from SPMs to MM through OPRs and SEN.

mI(105): activates nano and set MARs.
 [AM]

nIO(28): OPRO <- RFO(1)
 [EX]

```

mI(106): starts read MM operation; activates nano to receive the
          read data.
          [AA]

nIO(29): SPM0(1) <- IP0, ENP.

nI1(29): SPM1(1) <- IP1, ENP.

nI2(29): SPM2(1) <- IP2, ENP.

nI3(29): SPM3(1) <- IP3, ENP.

mI(107): activates nano and sets MARs.
          [AM]

nIO(2A): OPRO <- RFO(2), ENP.
          [EX]

mI(108): starts write MM operation; activates nano to send data
          to MM.
          [AA]

nIO(2B): OPRO <- SPM0(1), ENP.

nI1(2B): OPR1 <- SPM1(1), ENP.

nI2(2B): OPR2 <- SPM2(1), ENP.

nI3(2B): OPR3 <- SPM3(1), ENP.

```

Example 6: test branch Usual test branch operations are done by the combination of the TBN micro- and NTB nanoinstructions. The flow of the test results is shown in figure 3.17. The following is the parallel test of PU resources, i.e., DBUS=0 and counter CY=0, to make a microlevel branch:

```

mI(109): activates the same four nanoprogams to receive the test
          result through the TEST flags, and makes a microlevel
          branch. [TBN]

nIO(2C): TEST0 <- (DBUS0=0) AND (CY0=0)
          [NTB]

nI1(2C): TEST1 <- (DBUS1=0) AND (CY1=0)
          [NTB]

```

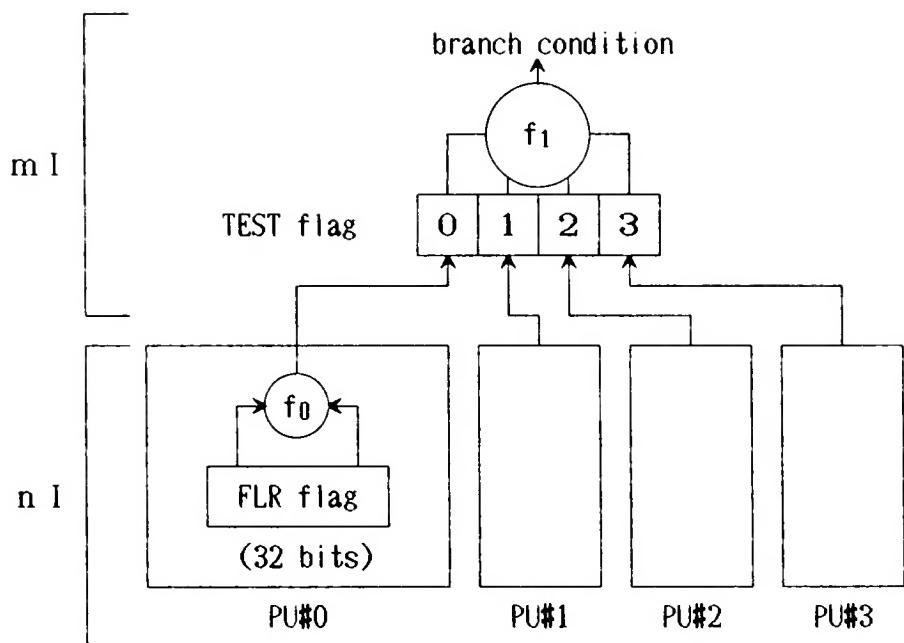


Figure 3.17
Parallel test.

nI2(2C): TEST2 \leftarrow (DBUS2=0) AND (CY2=0)
[NTB]

nI3(2C): TEST3 \leftarrow (DBUS3=0) AND (CY3=0)
[NTB]



4

Preliminary Evaluation

4.1 Application to Small Problems

The usual way to refine and evaluate a computer architecture designed is to simulate the execution of various programs or microprograms on the machine. The problems should be selected with care to ensure that the machine satisfies the principles of the design. The size of the problems should be convenient because usually we cannot perform good simulations at the initial phase of machine development. Further, we need a standard for determining relative design improvements.

Our selection for the preliminary evaluation emphasized nonnumeric problems, which are thought to be one of the weakest points of von Neumann machines (see chapter 1). The Data General ECLIPSE S/130 (hereafter referred to simply as ECLIPSE) minicomputer was chosen as the standard of comparison [DATA77]. The reason for this choice is that the ECLIPSE is user

microprogrammable and its microarchitecture is completely open to us. The microassembler helped us describe microprograms of a 56-bit horizontally structured microinstruction format. The parallelism by horizontal microinstruction, coupled with such nonnumeric data processing microoperations as bit set, byte exchange, and masking at the firmware level, provides firmware level users with a good environment for nonnumeric processing. To be fair, we compared the gate amounts of the machines; MUNAP's was about three times ECLIPSE's. Also, to ensure independence from the implementation technologies, we compared the results not by execution time but by the numbers of microsteps. The results are summarized in table 4.1.

1. **Bit processing:** For 256-bit data, three sample microprograms were described to count ones in the data, search the leftmost one, and set the specified bit. The ratios, from 4.9 to 36.3 in table 4.1, are due mainly to parallelism between the four processor units and bit operations by BOUs. Consolidation results of the four PUs is done in one step by using serial PU operation; this is evidence for the importance of speeding up interprocessor operations. If the serial PU operation were not so, it would require four steps even though the permutation function of the SEN is used to exchange data between PUs.
2. **Field handling:** Church's problem is a good benchmark for evaluating the field handling capability of a machine [CHUR70]. It transforms one data structure into another. Figure 4.1 shows the data structures before and after the transformation.

Table 4.1
Evaluation of nonnumeric functions

Problem		Ratio in microsteps	
		MUNAP	ECLIPSE
Bit processing	Bit count	1 (23) ^a	36.3
	Priority encode	1 (32)	25.6
	Bit set	1 (36)	4.9
Field handling	Church's problem1	1 (13)	4.2
	Church's problem2	1 (13)	3.9
Table access	8-bit table access	1 (87)	3.2
	16-bit table access	1 (398)	4.1
Variable length word accessing	8-bit word access	1 (2)	2.5
	32-bit word access	1 (2)	3.5
	128-bit word access	1 (4)	4.8
Sorting	Batcher's method	1 (1030)	3.4
	Counting	1 (1012)	3.6
Searching	Quick sequential	1 (261)	3.0
	Open hash	1 (260)	3.0

a. Numbers in parentheses indicate the execution steps of MUNAP microprograms.

LOCATION A	BITS 0	15 0	15
A+0	a (7)	b (9)	c (8)
A+2	d (8) →	e (8)	f (15)
A+4	h (12)	i (4)	j (10)
INPUT VARIABLES			
LOCATION B	BITS 0	15 0	15
B+0	g	d (16)	a (7)
B+2	f (7) →	b (9)	k (6)
B+4	h (12)	c (8)	e (8)
OUTPUT VARIABLES			
(a)			

BITS 0	15 0	15
LOCATION A	a (7) b (9) c (10) d (6)	A+1
A+2	e (15) f g (16)	A+3
A+4	h (12) i (4) j (8) k (8)	A+5
INPUT VARIABLES		
BITS 0	15 0	15
LOCATION B	g (16) b (9) d (6) 0	B+1
B+2	f 000 i (4) k (8) j (8) a (7) 0	B+3
B+4	0....0 c (10) 0 e (15)	B+5
B+6	0.....0 h (12) 0.....0	B+7
OUTPUT VARIABLES		

Figure 4.1
 Church's problem (C. C. Church 1970; © AFIPS 1970):
 (a) problem 1; (b) problem 2.

Comparing MUNAP with ECLIPSE having byte exchange and 8-bit mask generation functions, the ratios became 3.9 and 4.2. Further, the ratios, obtained by using HP2100 firmware [IIZU76], are 8.8 and 13.4 respectively. In MUNAP, the shift, divide and extract, stack, and concatenate functions of the DCUs as well as the exchange functions of the SEN contribute greatly to efficiency.

3. Table access: Table access is a basic function for nonnumeric processing. We defined the problem of scanning the words in the specified column, and retrieving rows if the values of the words equal the values of keys. The tables are 8 rows by 8 columns consisting of 8-bit words, or 16 rows by 16 columns consisting of 16-bit words. Their ratios are 3.2 and 4.1. This is because in MUNAP, 64-bit data may be read in parallel and concurrently, compared with the data in the four processor units. Further, the two-dimensional access function has proved to be quite useful for this kind of problem. Columnwise access of eight 8-bit data requires 2 steps in MUNAP, while it requires 43 steps in ECLIPSE. The 8-bit two-dimensional access function of MUNAP is utilized effectively for the table of 16-bit words.

4. Variable length word access: We defined the word size as being from 8 to 128 bits. The ratios are from 2.5 to 4.8. In MUNAP, the address mapping is performed by the AM hardware for word lengths from 8 to 64. The data read from eight banks of main memory are automatically adjusted by the SEN. On the other hand, the 16-bit fixed size access of ECLIPSE requires address mapping, extracting the necessary data field for smaller words,

and fetching several words for larger words.

5. Sorting: Sorting is one of the most important functions for nonnumeric processing. We chose two algorithms from Knuth's book [KNUT73], i.e., Batcher's parallel method and sorting by counting. These sort 32 16-bit data. Their ratios are 3.4 and 3.6. This is mainly due to parallelism among the four PUs, parallel data access from the eight-bank main memory, and permutation by the SEN. In particular, the regular permutation functions of the SEN, such as shift and data exchange, are quite useful for data transfer between multiple PUs and the main memory as well as exchange of data between the four PUs.

6. Searching: Searching, as well as sorting, are typical functions for nonnumeric applications. We chose a quick sequential search and hashing from among many searching algorithms [KNUT73]. For quick sequential search, the ratio is 3.0. In MUNAP, the key is broadcast to the four PUs by the SEN. Then the four words read from MM are compared with the key in parallel. The number of the processor unit in which the search succeeds is determined in one step by the microlevel functional branch according to the BOU priority encoded value of the test flags of four PUs in the direction from PU0 to PU3 (see figure 3.17). The ratio value of 3.0 for hashing is due to two facts: (1) scrambling functions may be easily implemented by using the DCU and SEN functions and (2) the search, which starts from a hashed address, may be done concurrently in four PUs.

We would like to point out one thing about the hashing. MUNAP can implement a parallel hashing method [IDA77] by generating different hashed addresses concurrently by using four different nanoprograms in the four PUs and accessing the 4 16-bit words in parallel by using the AM direct access mode. Although we used the same simple method in both MUNAP and ECLIPSE, the use of the parallel algorithm is a promising research topic for MUNAP.

4.2 Evaluation of Architecture

4.2.1 Evaluation Models and Existing Machines

The purpose of model definition is twofold: first, to evaluate the two-level microprogrammed multiprocessor architecture in a general frame, and second, to categorize the existing machines with architecture similar to MUNAP and position the MUNAP machine. **Figure 4.2** introduces the models.

Model A: This two-level microprogrammed multiprocessor model corresponds to MUNAP itself. Until now, we have had no other examples that strictly fit this type. However, some mainframe computers use the scheme for controlling several functional units. In HITAC M-200H, Hitachi's M-series general purpose computer [AIS080], the control units are distributed among the several functional units: General Arithmetic Unit (GU), Floating Point Arithmetic Unit (FU), Service Unit (SVU), Service Processor (SVP), and Input/Output Processor (IOP). This scheme contributes to speedup by shortening the distance between the control units and controlled units. The GU-FU decomposition is said to be most

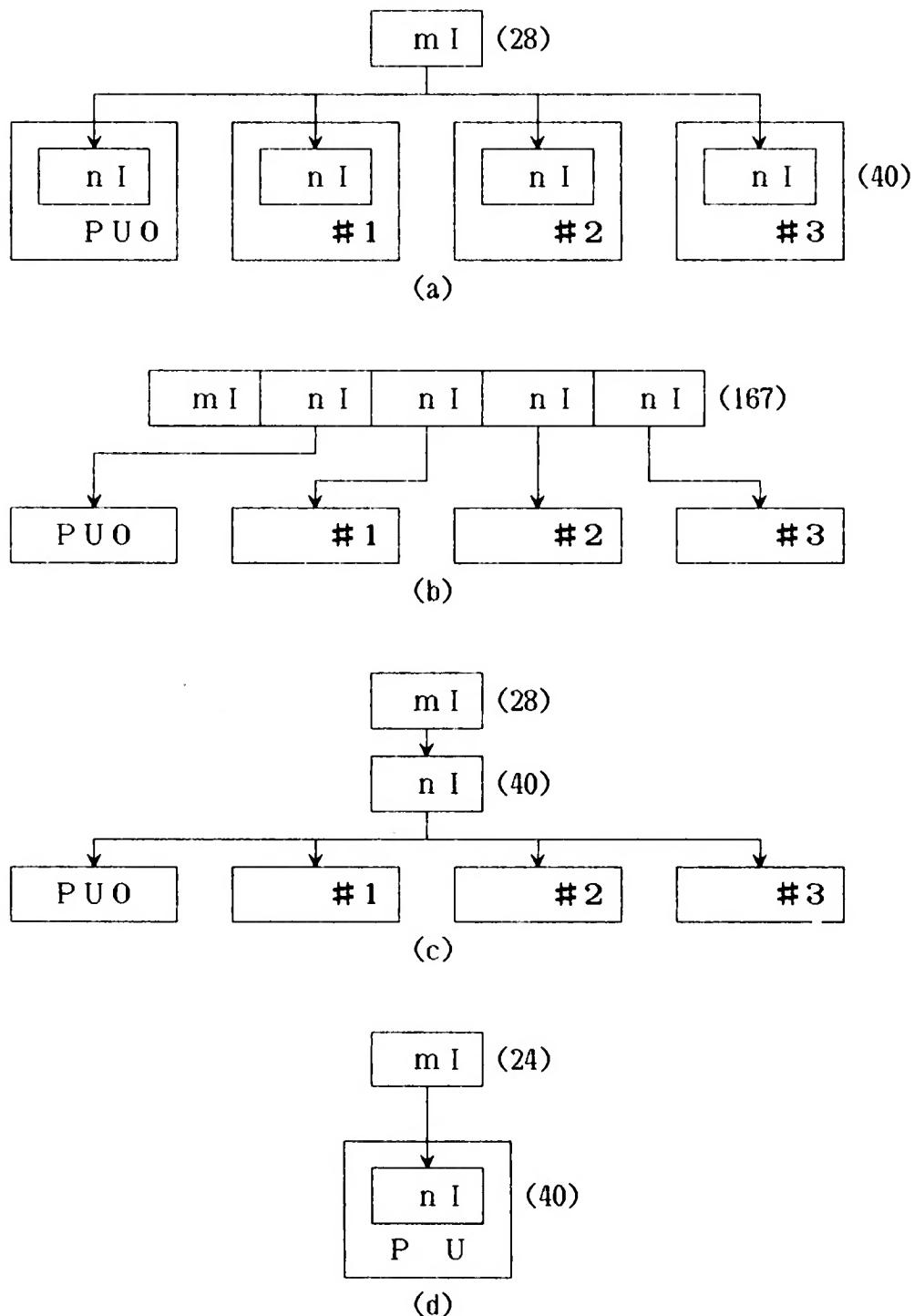


Figure 4.2
 Configuration of two-level microprogrammed processor
 (numbers in parentheses refer to instruction lengths):
 (a) model A: two-level multiprocessor (MUNAP);
 (b) model B: one-level multiprocessor; (c) model C:
 common nano, multiprocessor; (d) model D: uniprocessor.

effective for performance evaluation.

Model B: This is a one-level microprogrammed multiprocessor model. Each field of a microinstruction controls a processor or a functional unit. There are several commercial and experimental machines in this category. They feature a low level parallelism implemented by horizontally structured, relatively long microinstructions and parallel bus structure. To our best knowledge the **AMP** machine [BARR73], developed at Argonne National Laboratory, is the first to employ a very long (74-bit) microinstruction for controlling multiple ALUs. The **QA-1**, and **QA-2** are experimental machines, developed at Kyoto University, which are basically aimed at the utilization of the parallelism of four ALUs [HAGI80, TOMI83, TOMI86]. The Floating Point Systems **AP-120B** scientific array processor, which contains 10 functional units, such as adder and multiplier, is controlled by 64-bit instruction word [CHAR81]. The **ELI** machine employs an architecture with a very long instruction word, called **VLIW** [FISH83b, FISH84]. The so-called *trace scheduling* algorithm, a kind of global compaction technique, has been implemented in the compiler for automatically enhancing the parallelism.

Model C: This is the model for two-level microprogrammed multiprocessors with common nanoprogram. The effect of multiple nanoprogams of model A can be evaluated by comparing model A with this one. The control scheme of the **CM-1**, a prototype Connection Machine developed by Thinking Machines Corporation, seems to fall into this category. In this machine, the host

specifies higher level macroinstructions, which are interpreted by a so-called microcontroller to produce nanoinstructions for a large number of processor/memory cells [DANI85].

Model D: This model differs from the above three in that it employs a single processor architecture. Most of the two-level microprogrammed machines are in this category. The Burroughs Interpreter [REIG72] provides two types of 16-bit microinstructions and 54-bit nanoinstructions. The first type of microinstruction specifies a nanoinstruction address and initiates the nanooperations. The second type does not access nanoprogram memory, but rather directly controls microoperations. The Motorola MC68000 [STRI78] with 10-bit micro- and 70-bit nanoinstructions employs an approach similar to Interpreter. Figure 4.3 shows the internal organization of the MC68000 chip, which realizes complex instruction set architecture (CISC) using less control memory area than for a single-level control. The QM-1 decomposes a control specification into 16-bit (6-bit operation code and two 5-bit address parts) microinstructions and 342-bit (a 38-bit constant field and four 76-bit T-fields) nanoinstructions [ROSI72]. The microinstruction is executed by a nanoprogram, as an ordinary machine instruction is executed by a microprogram.

4.2.2 Evaluation

According to the model definitions, we modified the hardware organization and microinstruction formats of MUNAP, i.e. model A, to construct models B, C, and D. For fair comparison, the

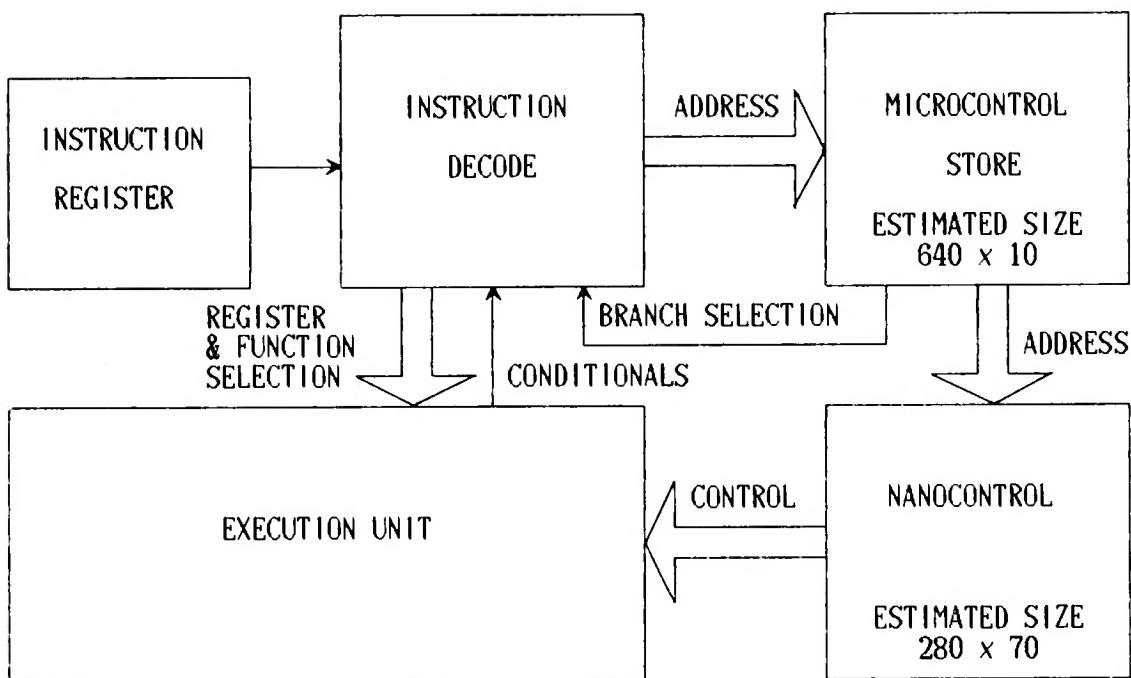


Figure 4.3
MC68000 control structure (S. Stritter and N. Tredennick 1978; © IEEE 1978).

modification was held to a minimum. Next, we selected four typical microprograms from those listed in table 4.1 and described microprograms for each model.

The results are shown in table 4.2. The amount of required control memory bits is calculated by adding the product of the microinstruction length and the number of microinstruction steps to the product of the nanoinstruction length and the number of nanoinstruction steps. The PU utilization represents an average number of simultaneously working PUs in one step.

a. **One-level versus two-level controls:** We first looked at models A and B. As they both have microlevel MIMD organization and consist of functionally equivalent multi-PUs, the number of execution steps and the PU utilization are the same. The reasons

Table 4.2
Evaluation of various configurations

Problem		A	B	C	D
Bit processing (bit count)	Microsteps	1 (23)*	1.0	1.3	3.6
	Control memory bits	1 (1048)	1.3	1.0	0.4
	PU utilization	2.8	2.8	2.3	1.0
Field handling (Church's problem 2)	Microsteps	1 (13)	1.0	2.6	2.8
	Control memory bits	1 (1976)	1.1	1.3	1.0
	PU utilization	3.2	3.2	1.2	1.0
Sorting (Batcher's method)	Microsteps	1 (1030)	1.0	1.5	3.2
	Control memory bits	1 (3244)	1.9	0.7	0.6
	PU utilization	2.1	2.1	1.1	1.0
Searching (Quick sequential)	Microsteps	1 (261)	1.0	1.3	3.9
	Control memory bits	1 (1336)	2.0	0.7	0.6
	PU utilization	3.0	3.0	2.5	1.0

a. The numbers in parentheses are real microsteps and control memory bits for configuration A, i.e., MUNAP.

why the ratios of the required control memory bits of B to A are 1.1 to 2.0 are as follows in machine A:

1. a microinstruction may activate a nanoprogam of several nanoinstructions;
2. a nanoprogam may be utilized among several microinstructions; and
3. the microinstruction, which does not control PUs (e.g., a sequencing microinstruction), does not activate nanoprogams in model A, while operation nanoinstructions are required for model

B.

In Batcher's parallel sorting microprogram, for example, (1) the average number of activated nanoinstructions per one microinstruction is 1.4.; (2) the percentage of nanoinstructions utilized by more than one microinstruction per the total number of nanoinstructions is 23%; and (3) the percentage of the microinstructions that do not activate nanoprogams per the total number of microinstructions is 8%. Further, advantages of model A over model B are that the modular functions in each PU may be described as nanoprogams and that, then, the frequently used functions may be provided to the user as a set of nanoprogams.

Further comparison, from table 4.2, may be made with respect to the complexity of decoding and the speed of execution. For decoding, if a microinstruction or nanoinstructions have equal flexibility in both models A and B, the decoding structure of A should be simpler than that of B, because the specification of nanoaddresses in model A is a kind of decoding that does not require hardware decoders. However, the present MUNAP has much more flexible relationships between micro- and nanoinstructions than that of single level control; the implementation of the relationships requires somewhat complex decoders to control successive execution of micro- and nanoinstructions. As to the speed of execution, two-level microprogramming may seem to cause overhead by fetching both micro- and nanoinstructions. In MUNAP, the reduction of overhead is realized by separating the execution of functions, and overlapping them as illustrated in figure 3.15.

b. **MIMD versus SIMD parallelisms:** In general, the possibility of parallelism by multiprocessors may be due to the following two reasons: (1) parallel processing for homogeneous data and (2) concurrent operations for heterogeneous data by detecting concurrency among different operations. The first reason corresponds to parallelism for the SIMD architecture; and the second to MIMD. The nanoprogram level MIMD organization allowed both parallelisms by using different nanoinstructions in multiprocessor units.

To clarify the effect of MIMD architecture by the distribution of nanoprogams, we compared models A and C. The execution steps for C are 1.3 to 2.6 times larger than for A. Model C can provides as much processing capability as model A for microprograms with homogeneous data structures such as bit count and searching. However, Church's problem 2 indicates that model C is not appropriate for processing heterogeneous data structures. The PU utilization also points up this fact. The ratios of the PU utilization of model C to that of model A are 0.9 for the bit count, but 0.4 for Church's problem 2. These indicate that the nanolevel SIMD architecture of model C cannot significantly improve the efficiency for heterogeneous data strcuctures. The ratios of the control memory bits of C to A are from 0.7 to 1.3. Thus, there are cases in which the increase in total execution steps for C requires more control bits than for A.

We also made a step-by-step analysis of two microprograms:

Batcher's sorting with homogeneous data and Church's problem 2 with heterogeneous data. For the Batcher, the SIMD steps (i.e., steps in each of which the same nanoinstructions are executed) are statically 32% and dynamically 72%; the MIMD steps are statically 18% and dynamically 19%. For the Church, the SIMD and MIMD steps are 15% and 85%, respectively. Since the Church microprogram is straight line, the static steps coincide with dynamic ones. This confirms that the uniformity of data and operations on them is reflected in the features of parallelism.

c. Single versus multiple processor units: We compared model A with model D. Model D requires 2.8 to 3.9 times as many execution steps as A. On the other hand, the control memory bits are 0.4 to 1.0 times less than for A. The amount of hardware for model D is approximately a fourth that for model A for PU, AM, and MM.

As it is true that we cannot expect a full utilization of multiprocessor parallelism, n processor units generally improves to m times less steps, where $m < n$. In the case for MUNAP, 4 times hardware realizes 2.8 to 3.9 times less steps. Although it is difficult to obtain m for n generally, our application to small problems shows around 3 for 4.

5.

Hardware Development

5.1 Hardware Development of an Experimental Machine

The computer architecture is essentially independent of its implementation. Standard books on computer architecture would not touch the details of implementation. Some proposed machines have never been implemented, especially in a laboratory environment. However, there are many reasons why, in our view, implementation is important:

- o The detailed design forces us to refine and clarify ambiguous parts of the architecture.
- o We can apply the machine to various areas considered suitable for the architecture and can examine the effect of

*Those who are not interested in implementation details can skip this chapter.

the architectural features on the computation. The evaluation based on the real application should become convincing.

Furthermore, we can learn much from this: how the architecture is broken down to a logical design; what kinds of ICs are available today and their speeds.

The emergence of large scale integrated circuits has benefited implementation. Large numbers of memory chips have been exchanged for small numbers of large capacity. Several vendors provide bit sliced chips for the functional units, such as the ALU, with register files. They also provide the microsequencer chips to be used with the functional units. Semiconductor manufacturers have set a pace in which commercially feasible chip complexity has doubled every one to two years. We can define the microinstruction formats at our convenience so as to add a new function or replace some parts with custom units. Thus, we can concentrate on realizing the architectural features, using the building blocks to make the basic frame of a machine. The situation is similar to VLSI (Very Large Scale Integrated circuit) processor design because many building blocks are available at the design phase.

In the following section we shall describe our case of machine implementation; it should be useful for architects planning to develop an experimental machine.

5.2 Procedure of Hardware Development

5.2.1 Development Process

After finishing the architecture design and hardware organization, we proceeded to the following phases.

1. Logical design: This phase is somewhat routine work that breaks down the hardware organization into primitive logical units, such as register, terminal, and AND/OR gates. The clock generator is also a primitive unit that provides timing signals to all the other parts.

2. Decomposition into ICs: This phase depends on the use of custom ICs. If we can design such ICs, this phase is an IC design that includes the functions at the logic design phase. If we use commercially available chips, this phase is the determination of appropriate chips that include necessary functions and function assignment. The requirements for the speed of the computation also affect the selection of ICs.

3. Assignment of the ICs to boards: The ICs should be mounted on boards. We can use printed boards that connect the ICs through printed connection circuits on the boards. If we use nonprinted boards, the connection is usually implemented by wire wrapping.

4. Connection between the boards: The connection between the boards may be a printed board, called a back panel, or a flat cable. As our laboratory environment did not allow us to design custom chips, we used commercially available chips. To make the

mounting process simpler, we mainly used gate ICs of LS type, control memory ICs of MOS type, and main memory ICs of dynamic RAMs. The connection was mainly by wire wrapping. However, flat cables were used to connect macro modules.

Throughout the development, we tried to leave room for future correction and extension, utilizing the benefits of microprogrammed control. The extensible designs are a set of micro- and nanoinstructions, their fields, and the micro- and nanoorders. The boards have ample room for mounting new ICs that are not included in the initial design. The connectors have undefined signal lines to be used later. These have proved to be effective for the correction and extension of an experimental machine, especially when it employs a microprogramming scheme.

5.2.2 Mounting

The microlevel was decomposed into two blocks, as shown in figures 5.1 and 5.2. The diagrams associated with the pictures illustrate the modules that make a block. Each block consists of 12 boards, and each board contains up to 50 ICs. The first block includes the interface with the host, microregisters, address modifier, and main memory. The second block contains the microsequencer, microprogram memory, micro-nano flags (MNFL) and its MASK, microinstruction decoder, microstacks, and shuffle exchange network. The total amount of microlevel ICs is 761 (356 SSI, 293 MSI, and 112 LSI). They consume about 85 W.

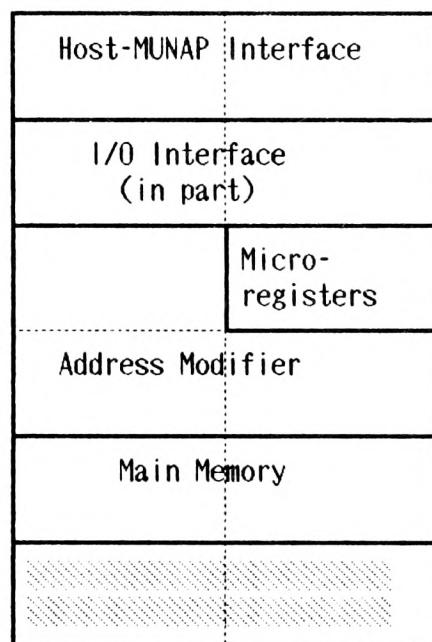
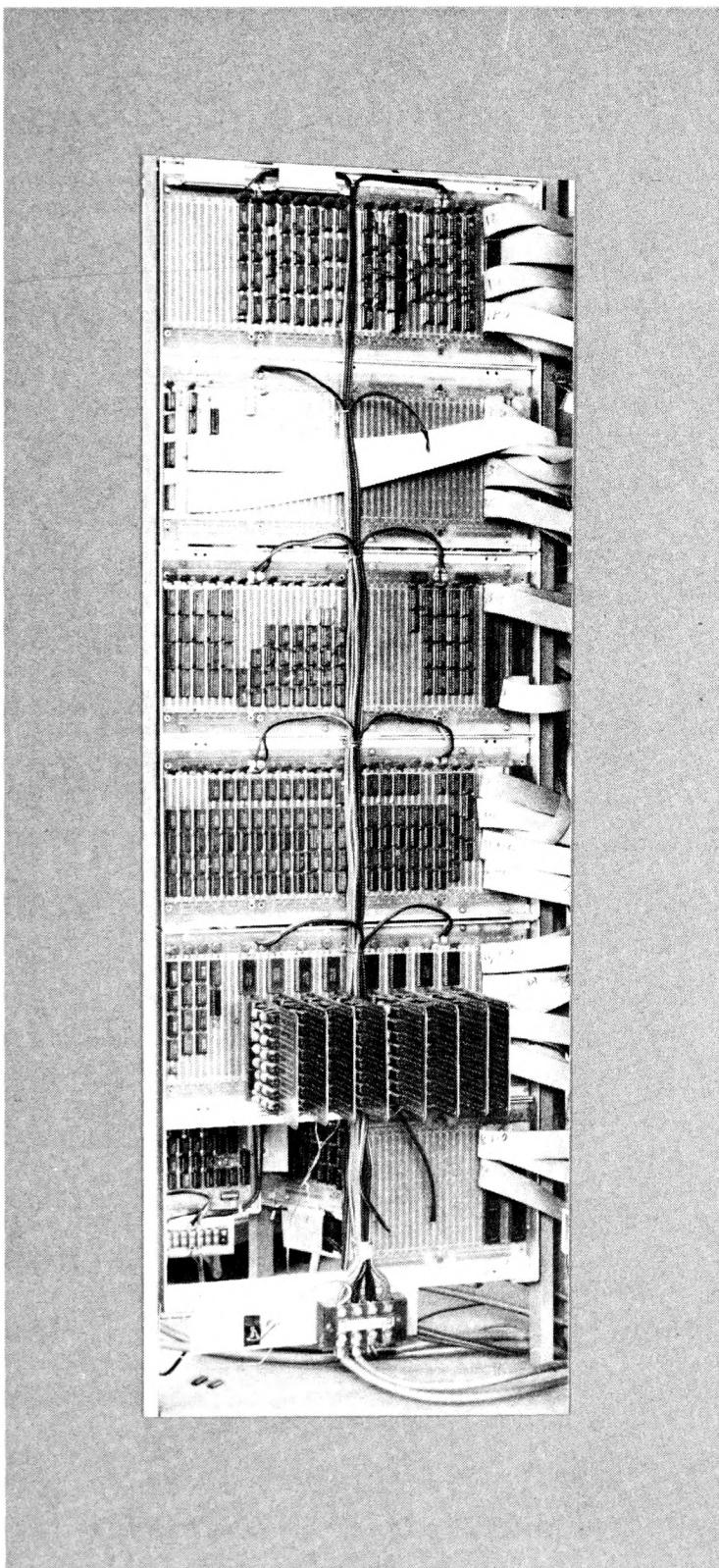


Figure 5.1
Microblock 1.

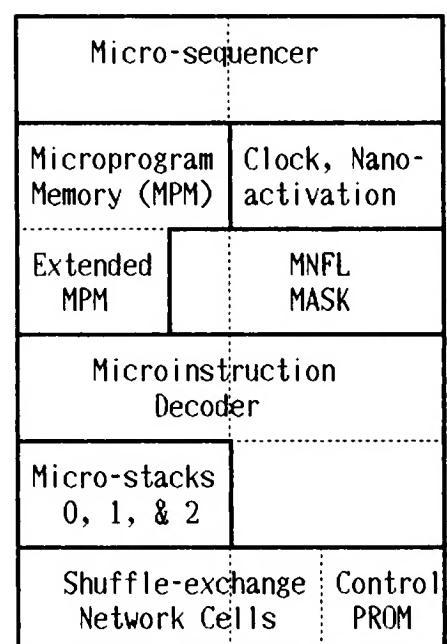
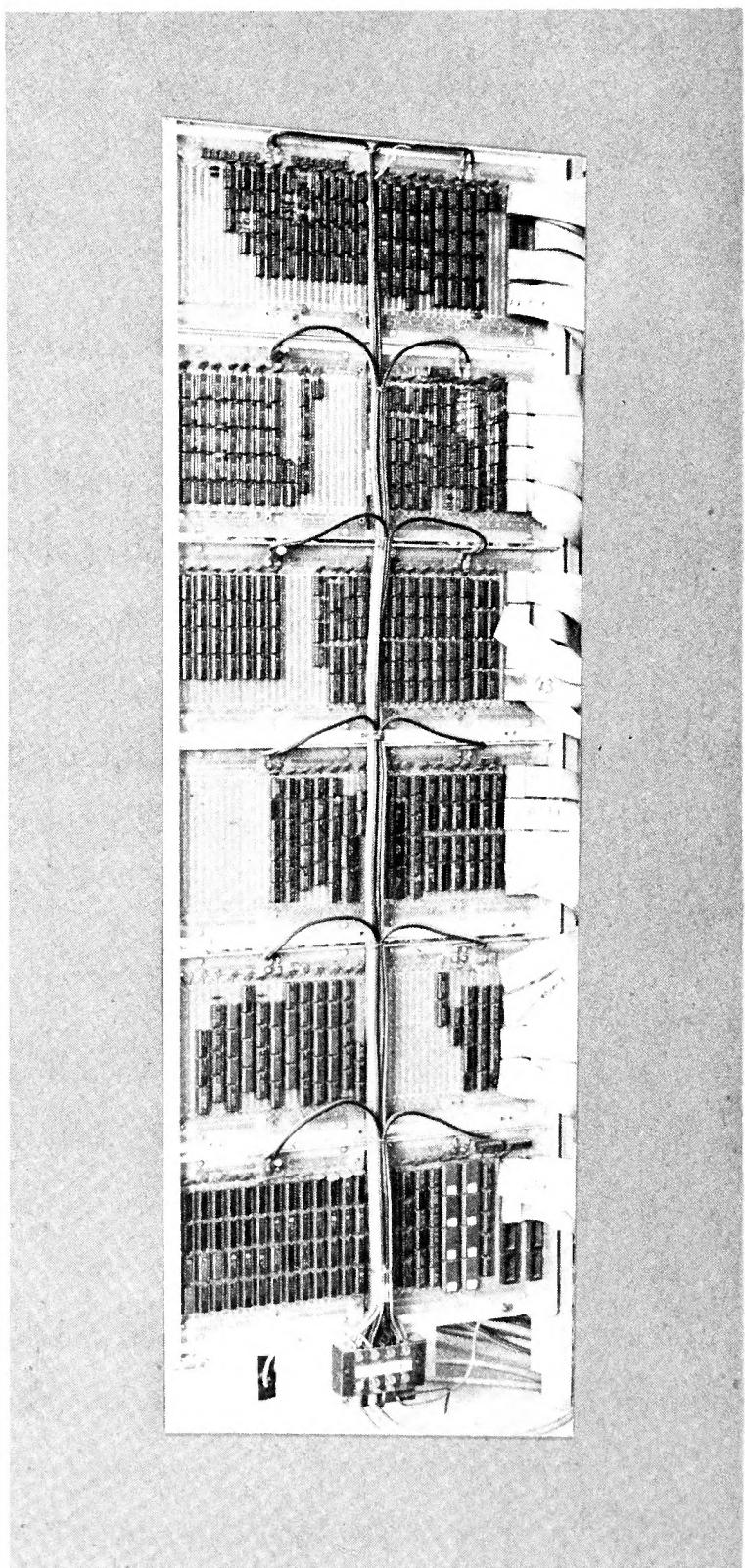


Figure 5.2
Microblock 2.

The four processor units correspond to four blocks, each consisting of 9 boards, as shown in figure 5.3. This includes the nanoprogram memory, nanosequencer, nanoinstruction decoder, ALU, DCU, BOU, scratchpad memory, counter, flags, and port registers. Each processor unit block consists of 422 ICs (212 SSI, 143 MSI, 67 LSI). They consume about 39 W.

Figure 5.4 shows a general view of the MUNAP prototype, made of the 2 micro- and 4 nanoblocks. The MUNAP itself consists of about 2,500 ICs and consumes about 240 W. By adding hardware for the interface with the host and I/O devices, the present MUNAP consists of about 3,000 ICs and consumes about 280 W.

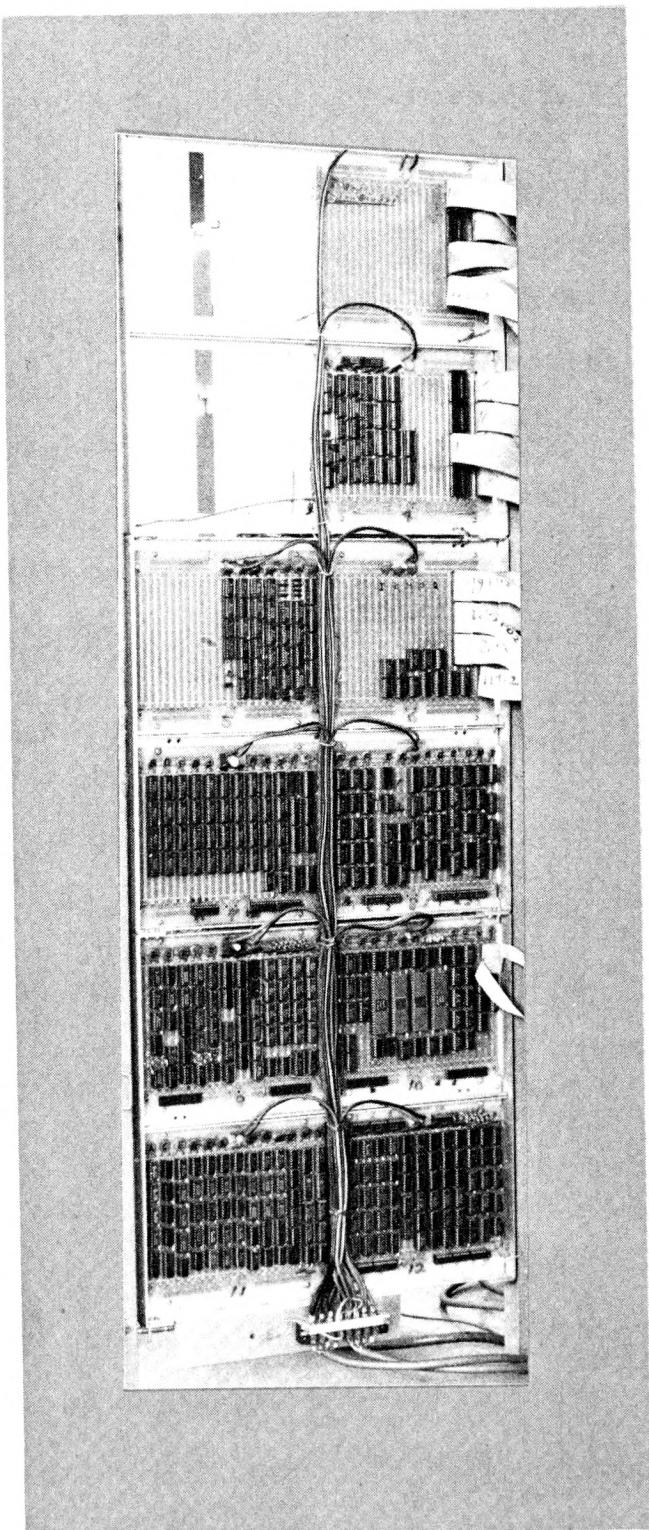
Within a block, ICs were connected by wire wrapping. Flat cables were used to connect the blocks.

5.2.3 Documentation

Throughout the development, we kept in mind the importance of documentation, so that the description for each block contains a summary of functions, a block diagram of functional units, detailed logical design, allocation to boards, and a list of wire wrappings. This documentation facilitated later work on maintenance and extension.

5.3 Interface with the Other Systems

MUNAP is not a stand alone computer. As shown in figure 5.5, it is connected to the other computer systems and I/O devices with the following objectives.



		Port Registers	
	Nano-Sequencer		Clock Buffer
Nanoprogram Memory		Nano-decoder & Flag reg.	
Counter & SPM		ALU	
Bit Opera. Unit (BOU)		Divide&Concat. Unit (DCU)	

Figure 5.3
Processor unit block.

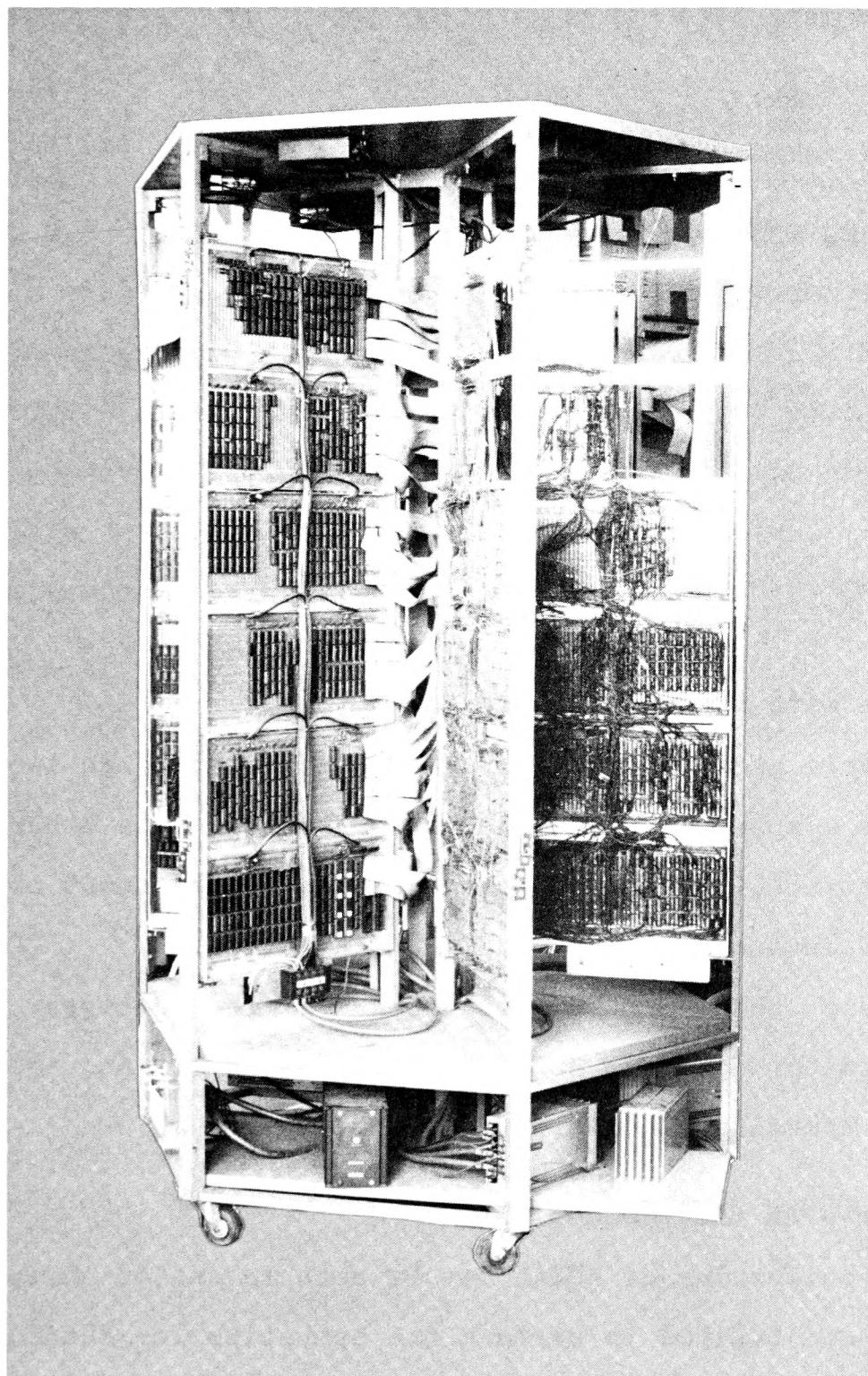


Figure 5.4
General view of MUNAP.

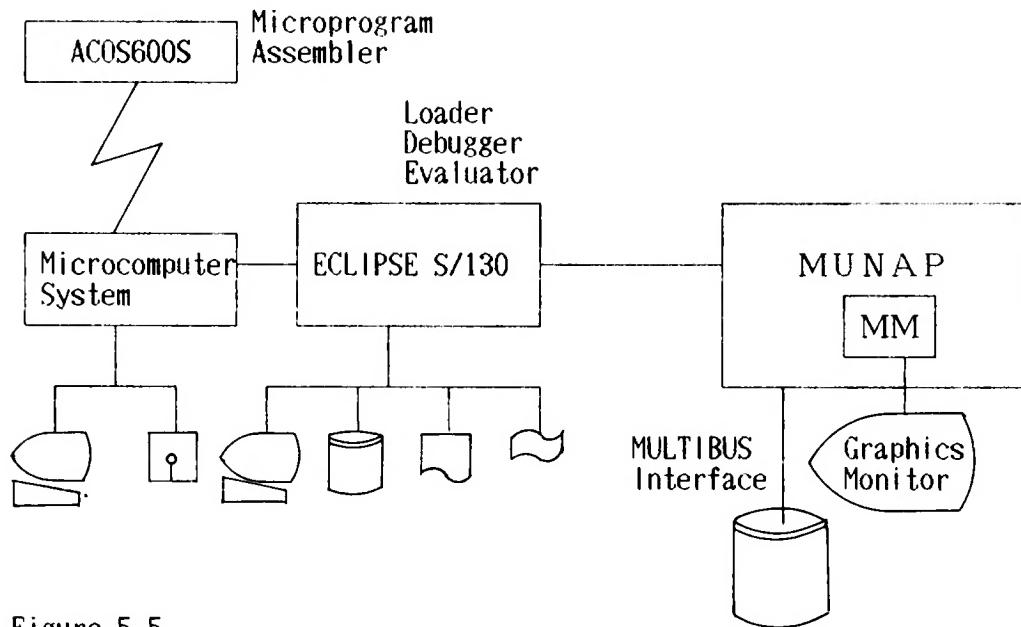


Figure 5.5
Computer network for MUNAP.

5.3.1 Interaction with the Console Processor

MUNAP and the console processor ECLIPSE are connected through two bus lines. One is a special purpose bus between them. The other is a MULTIBUS interface, which allows an arbitrary attachment of a computer and I/O devices with the MULTIBUS interface.

These buses are used for I/O operations of the debugger, loader, and evaluator on the ECLIPSE (intermediate language transfer from the translators developed on ECLIPSE).

5.3.2 Interaction with the I/O Devices

The inputting and outputting of MUNAP may be done in one of three ways. One is to ask ECLIPSE to perform the operation for MUNAP. This is easy but slow. Some applications require direct and fast operations. The second is to use the MULTIBUS interface. Motorola MC68000 and disk hang off from the bus. The third is a direct connection to MUNAP. A graphic monitor is an example of

this type. It is connected to MUNAP through a common main memory attached to MUNAP for the connection. The special hardware realizes fast I/O operations.

5.3.3 Interaction with a Large Computer System

As the cross softwares, such as a microassembler and a system description language translator, had been developed on the ACOS 600S, a relatively large scale computer, we needed to transfer the object microprograms and intermediate programs to MUNAP. This was realized by a transfer system, developed on a microcomputer system, as shown in figure 5.5.



6

FIRMWARE ENGINEERING

6.1 Firmware Engineering

As the increase in software development cost was the motivation for software engineering, the need for a large amount of microprogram development is now motivating new technologies, so-called *firmware engineering* [ANDR80]. It is supposed to include the description, translation (optimization), and verification of microprograms. Unlike software engineering, firmware engineering is still in its infancy. Many manufacturers have developed low level microprogramming languages. Sometimes, the language consists of lists of microorder mnemonics with labels for symbolic addressing. The sources of the difficulty for firmware engineering are as follows:

- a. **The need for efficient object codes:** The efficiency of microprograms is very important because they are the basis of software and are to be used repeatedly. The compiled microprograms are less efficient than hand coded ones.
- b. **Many constraints from hardware features:** Firmware suffers many constraints from its underlying hardware, such as timing and resource conflicts. Firmware engineers who are experts, sometimes use tricky codings to maximize concurrency in the face of those constraints.
- c. **The small sizes of microprograms:** The sizes of microprograms are usually not as large as programs for a large software system, which can be handled without the help from a support system.
- d. **Difficulty of machine independence:** The hardware features, described in (b), differ from one machine to another. Thus, it is very difficult to develop machine independent technologies that can be applied to every machine and can produce an efficient object code.

In addition to these difficulties, the more innovative the microarchitecture is, the more difficult the development of the support system. In our case, the difficulty involves a two-level microprogramming scheme coupled with multiprocessor parallelism. The work of many researchers on firmware engineering has yielded remarkable results [BABA81b]. These results may be classified in the following three categories: description, translation, and verification. In particular, special optimization technologies

have been developed for exploiting microarchitecture parallelism at the translation phase.

In this section, we shall describe these results and clarify the framework of our firmware development system [BABA82a].

6.2 Description of Two-Level Microprograms

6.2.1 Design Objectives

Our microprogramming language is a register transfer level one with machine dependent features. The language allows the user to utilize fully the characteristics of the machine. The language features that facilitate the description of two-level microprograms with multiprocessor parallelism are summed up as follows.

1. **Virtual Microarchitecture:** Basically, the user can describe any combination of a microinstruction and multinanoprograms, activated by the microinstruction, in order efficiently to make use of the flexibility of two-level microprograms. The microprogram and nanoprograms are described in a sequence and distinguished by indentation. At this level, the hardware functions are uniform and frequently used functions may be described in one statement without being aware of the operations of elementary micro- and nanoinstructions. The following examples illustrate the uniformity and one statement description, respectively.

Example 1: XIF POS2=1 THEN GOTO LO;

Really, the positive flag (POS) does not exist. This is to avoid the redundancy of hardware that has test functions for zero and negative data. However, this causes inconvenience for the user. The virtual flag, named POS, is defined to make the test functions uniform.

Example 2: XIF FX1=0 THEN GOTO LO;

Similar to example 1. FX1=0 can be tested by this statement, which is not directly implemented in hardware.

Example 3: SPM0,1(5) := RF2,3(6);

Two parallel data streams are written in one statement. The contents of register file RF(6) (6 represents the address) are read in parallel from processor units (PUs) 2 and 3, and sent to SPM(5) in PUs 0 and 1 through the shuffle exchange network (SEN). As will be described later, the data reading and writing at the PUs are controlled by nanoinstructions and the data transfers between PUs are controlled by a microinstruction. The SEN shift amount is automatically determined by the language processor.

2. Description of Parallel Processing: The parallel processing of MUNAP is divided into three categories: (i) microinstruction and multiple nanoinstructions, which are tightly coupled to perform a single task; (ii) the SIMD operation, in which multiple

PUs do the same operation; and (iii) the MIMD operation. Example 3 is an example of (i). Although the MIMD operation should be described in several statements, the SIMD operation may be described in one statement. Examples 4 and 5 show the examples for SIMD and MIMD operations, respectively:

Example 4: SPMO,1,2(A) := RFO,1,2(B) <+> 4;

Three parallel additions are performed in PUO, 1, and 2 by three nanoinstructions.

Example 5: SPMO(A) := RFO(B) <+> 1;
 SPMO(A) := RFO(B) <+> 2;
 SPMO(A) := RFO(B) <+> 3;

If the operations are different, they are described separately.

3. Description of Nonnumeric Functions: One of the major features of MUNAP is its rich nonnumeric functions, such as logical address by the address modifier, the data exchange and broadcast by the shuffle exchange network, and bit and field operations by the BOU and DCU. To facilitate use, many operators were defined to represent the functions. They are indicated by angle brackets. For example, <+>, <BCT,1>, and <SRL16> represent addition by the ALU, count 1's by BOU, and 16-bit shift by the SEN, respectively.

6.2.2 Language Syntax and an Example Description

The microprogram consists of one main and multiple subblocks.

Each block is divided into declaration and execution parts.

Table 6.1 shows the statements that are included in each block.

There are three declaration statements. Executional statements

Table 6.1
Microassembly language statements

Statement name (meaning) [micro- and nanoinstructions]	
Declaration	EXT (External Address) ADDR (Internal Address) EQU (Name for an SPM or RF word)
Execution	
Micro (activate nano)	Assignment (including Read/Write MM)[AA,AB,BA,CB], MPM Write [MPMW], AM (MM Address Modification) [AM], *GOTO [BN], *IF (2-way/5-way branch) [TBN],
Micro (not activate nano)	GOTO, IF, CASE (Functional Branch), CALL, RETURN [B], Literal [LT], Flag SET, RESET [MNFC], IF (2-way/5-way branch) [TB],
Nano	ALU [ALU], DCU [DCU], BOU [BOU], Nano-literal [NLT], IF [NTB], SET/RESET MNFL, MICRO (Data transfer to Port Register) [EX], NOP [NOP]
Micro Nano combined	Assignment (Data Transfer between Micro and Nano) [Combinations of micro-AA, AB, BA,CB and nano- ALU, DCU, BOU, NLT, EX], *IF (Microlevel Branch Based on Nanolevel Test Results) [micro-TBN and nano-NTB]

have their corresponding micro- or nanoinstructions, which are indicated by [and]. They are further divided into three groups: microstatements, nanostatements, and micro-nanostatements. Some microstatements activate nanoprogams, and others do not. The nanostatements describe the operation in the processor units. The micro-nanocombined statements describe the combined operations of a microinstruction and the activated nanoprogams.

Figure 6.1 is an example description of the bit count microprogram as described in section 4.1. In general, the micro and nano are distinguished by indentation. For example, ST-NO. 3 describes the microinstruction, which activates the nanoinstructions of ST-NOs. 4 and 5. The symbol * attached to

ST-NO. SOURCE STATEMENT

```

1 MICRO MAIN BIT COUNT (100);          /* BIT COUNT MICROPROGRAM */
2 EXT NEXT (2);                      /* DECLARE EXTERNAL SYMBOL NEXT */
3   *;                                /* ACTIVATE NANOPROGRAMS */
4     RF(2) := 0;                      /* CLEAR BIT COUNTER */
5     CX0 := 0;                        /* CLEAR COUNTER CX0 */
6 L1: AM MODE M8 (X,H) PU(3-0);      /* SET MM ADDRESS WITH SUM OF
7       OPR := RF0(1) <+> CX0;        RF0(1) AND CX0 */
8     SPM(10) := MM;                  /* READ DATA INTO SCRATCHPAD */
9     RF(3) := <BCT,1> SPM(10);      /* COUNT ONES IN SPM(10)'S */
10    RF(2) := RF(2) <+> RF(3) CX0+1; /* COMPUTE PARTIAL SUM */
11    *IF CX0 MOD4 <> 0 THEN GOTO L1; /* TEST COUNTER */
12    IPR := <SRL16> OPR;           /* ADD RF0(2)-RF3(2) AND SET
13      OPR0 := RF0(2);              RF3(4) BY SERIAL PU
14      OPR1 := RF1(2) <+> IPR1;    OPERATION */
15      OPR2 := RF2(2) <+> IPR2;
16      RF3(4) := RF3(2) <+> IPR3;
17    GOTO NEXT;
18 END;

```

Figure 6.1
An example bit count microprogram.

the ST-NO.3 represents that the microinstruction activates the succeeding nanoinstructions. The fourth statement is an example of a one-statement description of an SIMD operation that simultaneously clears the second words of four register files in the four processor units. The fifth statement clears the counter CXO in PUO. Statements 6 and 7 are coupled to set the address of the main memory in the 8-byte normal mode (indicated by M8). Statements from 8 to 10 count the number of ones in the 64-bit data. And statements from 12 to 16 add the results in four processor units by the serial PU operation as shown in figure 3.13b.

6.3 Microassembler and Nanoprogram Optimization

6.3.1 System Configuration

The major features of the translation are the followings.

1. decompose a micro-nano combined statement into a microinstruction and nanoinstructions, and assign the nanoinstructions to appropriate PUs;
2. extend a statement for multiple PUs to several statements and assign them to appropriate PUs; and
3. optimize the two-level microprograms.

Figure 6.2 represents the system configuration of the microassembler. It consists of three phases. To leave room for extension, some microinstruction formats and microorders in

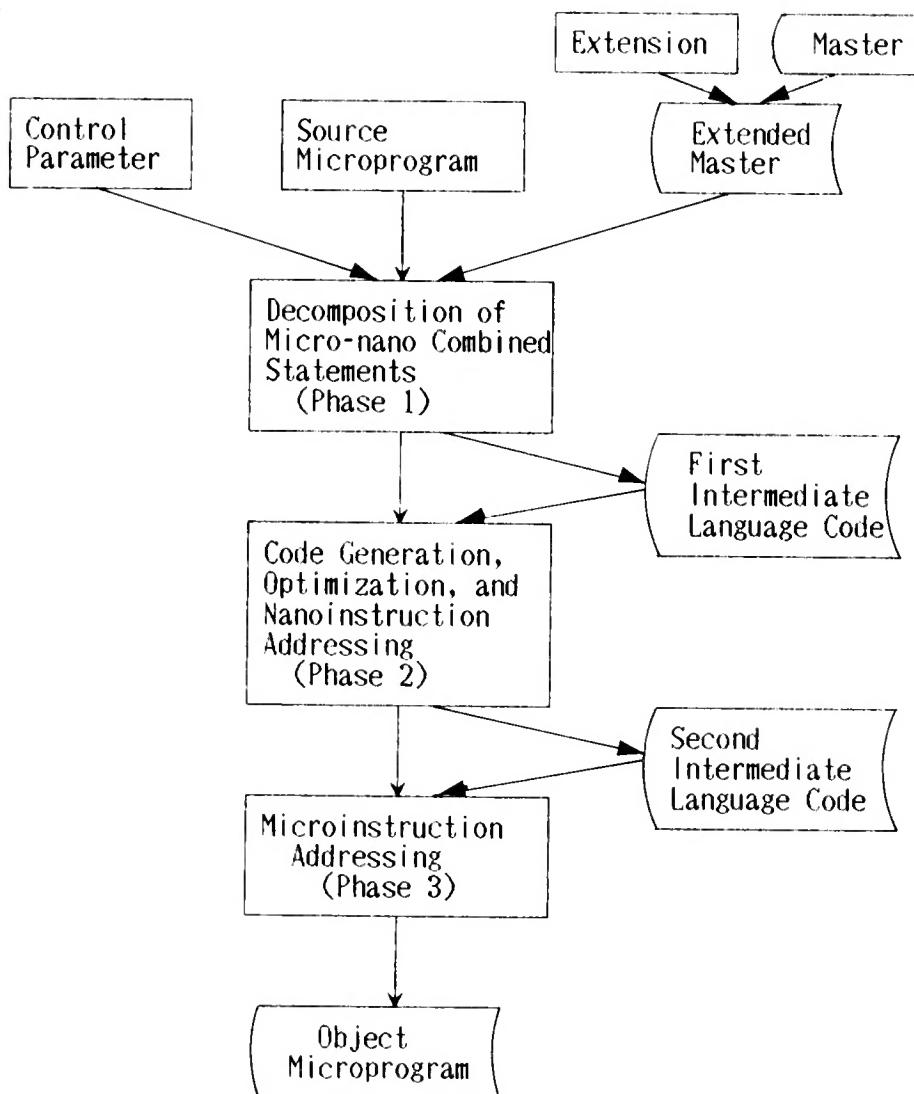


Figure 6.2
System configuration of microassembler.

microinstruction fields are left undefined. They may be defined by inputting the extension to the master file. This is particularly useful for an experimental machine, which may be extended after the initial development.

The system had been developed using 6,200 PL/I statements on ACOS 600S. The object codes may be transferred to ECLIPSE S/130, the console processor of MUNAP.

6.3.2 Decomposition into Micro- and Nanoinstructions

At the first phase a described source microprogram is decomposed into microinstructions and their nanoprograms. If a statement corresponds to a microinstruction or a nanoinstruction, the task is trivial. However, a micro-nano combined statement should be decomposed into a microinstruction and the nanoinstructions. A parallel nanostatement should be decomposed into identical nanoinstructions. The following are examples of decomposition.

1. **Assignment statements:** Among the assignment statements, (a) parallel nanostatements and (b) micro-nano combined statements should be decomposed. The statements of example 4 and example 3 are the examples of (a) and (b), respectively. In the following decompositions, m and ni represent a microinstruction and a nanoinstruction of PU# i , respectively. A register transfer form is used to represent the contents of object codes, instead of bit string form.

(Decomposition of example 4)

```
n0: RFO(B) <+> 4 -> SPM0(A)
n1: RF1(B) <+> 4 -> SPM1(A)
n2: RF2(B) <+> 4 -> SPM2(A)
```

(Decomposition of example 3)

$m: OPR \rightarrow \langle 32\text{-bit shift by the SEN} \rangle \rightarrow IPR$	
$n2: RF2(6) \rightarrow OPR2$	$n0: IPR0 \rightarrow SPM0(5)$
$n3: RF3(6) \rightarrow OPR3$	$n1: IPR1 \rightarrow SPM1(5)$

IPR_i and OPR_i are the input and output port registers of PU# i , respectively. As shown in figure 3.16, the read operations at PU2 and 3 are controlled by n_2 and n_3 . The transfer from the $OPR_{2,3}$ to $IPR_{0,1}$ through the SEN is controlled by m . The write operation at PU0,1 is controlled by n_0 and n_1 .

2. Test branch: The mechanism for reflecting the nanolevel test result to microlevel is depicted in figure 3.17. Each PU selects two bits from a 32-bit FLR flag, applies one of three logical operations ('or', 'and', and 'exclusive or') on them, and sets the result on the TEST flag. The microinstruction further specifies one of three logical operations ('or', 'and', and 'priority encoding') and the result is used for microlevel branch. Thus, using the mechanism, example 1 and example 2 can be decomposed as follows.

(Decomposition of example 1)

```

m: IF TEST2=1 THEN GOTO LO
n2: IF NEG2=1 NOR DZ2=1 THEN SET TEST2

```

The not-or (NOR) operation for negative (NEG) and zero (DZ) realizes virtual positive (POS) flag.

(Decomposition of example 2)

```

m: IF TEST1=1 THEN GOTO L1
n1: IF FX1=1 NAND FX1=1 THEN SET TEST1

```

The not-and (NAND) operation for two test conditions $FX_1=1$ realizes $FX_1=0$.

Notice that these combinations of micro- and nanoinstructions may be executed in parallel in one machine cycle. The decomposed micro- and nanoinstructions are to be passed to phase 2 with the symbol table for the both levels and the statement number table.

6.3.3 Nanoprogam Optimization

The optimization process of the microassembler is divided into two phases, namely, nanoprogam utilization and nanoaddress space compaction.

1. Nanoprogam utilization (optimization I): Let $mI(A)$ and $mI(B)$ be the microinstructions of addresses A and B, respectively. Let $\{PA\}$ and $\{PB\}$ be the sets of PU numbers activated by $mI(A)$ and $mI(B)$, respectively. $\{nA(i,j)\}$ and $\{nB(i,j)\}$ (i : PU number, j : nanoinstruction number) are the sets of nanoinstructions activated by $mI(A)$ and $mI(B)$, respectively. Then, if the following condition is satisfied, $mI(A)$ can utilize a part of the nanoinstructions $\{nB(i,j)\}$ for $mI(B)$.

Condition 6.1: $\{PA\} \subseteq \{PB\}$ and $\exists k$ (k : nanonegative integer) such that $\forall p \in \{PA\}$, $nA(p,j) = nB(p,j+k)$ ($j = 0, 1, \dots, SAp - 1$), where SAp is the number of steps of $\{nA(p,j)\}$.

In this case, if $mI(B)$ specifies nanoaddress b, then $mI(A)$ should specify $b+k$. The check using condition 1 requires a large number of comparisons between nanoprogams. To reduce the computation time, we have defined a necessary condition for condition 6.1

based on the fact that the ENP field of the last nanoinstruction of a nanoprogram should be 1 and that of the other nanoinstructions should be 0.

Condition 6.2: If condition 6.1 is satisfied, then $\forall p \in \{PA\}$, $\exists h$ (h: nonnegative integer) such that $SB_p = SA_p + h$, where SA_p and SB_p are the numbers of nanoinstructions for nanoprogams $\{nA(p, j)\}$ and $\{nB(p, j)\}$, respectively.

Figure 6.3 shows an example. We assume that $nA(0,0) = nB(0,2)$, $nA(0,1) = nB(0,3)$, ..., $nA(3,1) = nB(3,3)$. Then, condition 6.1 is

		PU#0	PU#1	PU#2	PU#3
$mI(A)$ $\{PA\}=\{0,2,3\}$	a :	nA (0,0)		nA (2,0)	nA (3,0)
	a+1 :	nA (0,1)			nA (3,1)
	a+2 :	nA (0,2)			
		SA ₀ =3	SA ₁ =0	SA ₂ =1	SA ₃ =2
$mI(B)$ $\{PB\}=\{0,1,2,3\}$	b :	nB (0,0)	nB (1,0)	nB (2,0)	nB (3,0)
	b+1 :	nB (0,1)	nB (1,1)	nB (2,1)	nB (3,1)
	b+2 :	nB (0,2)		nB (2,2)	nB (3,2)
	b+3 :	nB (0,3)			nB (3,3)
	b+4 :	nB (0,4)			
		SB ₀ =5	SB ₁ =2	SB ₂ =3	SB ₃ =4

Figure 6.3
An example of nanoprogram utilization.

satisfied and $k = 2$. Clearly, condition 6.2 is satisfied and $h = 2$. Thus, the shaded area of nanoprogram nB may be used from $mI(A)$ just by changing its nanoaddress specification field from a to $b+2$.

2. Nanoaddress space compaction (optimization II): Let sp be the nanoprogram steps of $PU\#p$. Let $\{V(p,k)\}$ ($p=0, 1, 2, 3, k$: nonnegative integer) represent a set of nanoaddresses of $PU\#p$. The condition for the nanocode movement is given as follows:

Condition 6.3: $\forall p \in \{P\}, \exists j$ such that $V(p,j), V(p,j+1), \dots, V(p,j+sp-1)$ are unassigned nanomemory addresses for $PU\#p$.

The meaning of this condition will be clear from figure 6.4. In figure 6.4a, each row and column indicate nanoaddress and processor unit, respectively. The same numbers, written in the tables, identify nanoprogams activated by a microinstruction with the specified number. For example, $mI(0)$ activates nanoprogams in PUs 0, 1, and 3 with nano start address 100. According to the condition, the nanoaddress space is compacted as shown in figure 6.4b. The condition is applied to microprograms that are optimized by optimization I. According to the direction of the code movement, there are two kinds of compactions, namely, forward and backward ones, which try to move a nanoprogam to a space with small and large addresses, respectively.

The sequencing microorders are added to the optimized microprogram. Then the microprogram is encoded to produce an

Nanoaddress	Processor Unit				Nanoaddress	Processor Unit			
	0	1	2	3		0	1	2	3
100	0	0	0		100	0	0	4	0
101	0				101	0	1	1	
102		1	1		102	2		2	
103	2			2	103	3	3	3	2
104				2	104				
105	3	3	3		105				
106			4		106				

(a)

(b)

Figure 6.4

Nanoaddress space compaction: (a) before compaction; (b) after compaction.

output from the microassembler. Figures 6.5 and 6.6 illustrate, respectively, microprogram and nanoprogram (only for PUO) object for the bit count microprogram shown in figure 6.1. The list for each instruction consists of the corresponding source statement number (ST-NO.), allocated micro- or nanoaddress, mnemonics, and binary/hexadecimal representations. The microassembler also produces other listings for the user. The example is the control memory map, as shown in figure 6.7, which indicates the relationship between a microinstruction and the corresponding nanoprogram. For example, it shows that the microinstruction at address 100 activates four nanoprogams in PUO,1,2, and 3 starting from nanoaddress 100. The reader is encouraged to check the correspondence between the source program in figure 6.1 with the object microcodes.

ST-N0.	MICRO ADDRESS	MICRO OBJECT	HEX DECIMAL
3	100		
BN SMA=1 NXR=0 NA=100 PU=0,1,2,3	1100 0000001 0 000100000000 1111	C02100F	
6	101		
AM OBO M8 X H NXR=0 NA=102 PU=0	1011 00 011 1 0 0 000100000010 1000	B1C1028	
8	102		
AA SLO MM PU NXR=0 NA=103 PU=0,1,2,3	00 00000 01 00 0 000100000011 1111	008103F	
11	103		
TBN AND TMA=-2 NXR=0 NA=106 PU=0	11101 00 1110 0 000100000110 1000	E9C1068	
12	104		
AA SL48 PU PU NXR=0 NA=107 PU=0,1,2,3	00 01100 00 00 0 000100000111 1111	180107F	
17	105		
B 0B3 P=0 MAD=0002 NOP	11010 11 0 0000000000000010 0000	D600020	

Figure 6.5
Microprogram object code.

ST-NO.	NANO ADDRESS	NANO OBJECT	HEX DECIMAL
4	100		
	NLT NLT=0000 RFB2 NOP ENP=0		
	1101 00000 0000000000000000 100000010 00000 0		D000004080
5	101		
	NLT NLT=0000 CX NOP ENP=1		
	1101 00000 0000000000000000 100100010 00000 1		D000004881
7	102		
	ALU ADD SH4 AC0=0 RFA1 CX DOPR NOP ENP=1		
	0 10011 00100 0 0001 1400100010 100110111 00000 1		4C81914DC1
8	103		
	EX EXT NOP SPM(10) NOP ENP=0		
	1100 00000000001 100011111 000010000 00000 0		C0018F8400
9	104		
	BOU BCT COUNT1 NOP SPM(10) RFB3 NOP ENP=0		
	101 011 00000 10000 000010000 100000011 00000 0		AC100840C0
10	105		
	ALU ADD SH4 AC0=0 RFA2 RFB3 RFB2 UPX ENP=1		
	0 10011 00100 0 00101 100000011 1000000101 01000 1		4C8281C091
11	106		
	NTB NOR X4 X4 NA=000 NOP ENP=1		
	1110 00000 011 11011 000000000000 00000 1		E03DEC0001
13	107		
	ALU NOR SH4 AC0=0 RFA2 NOP DOPR NOP ENP=1		
	0 10110 00100 0 0010 100011111 100110111 00000 1		58828FCDC1

Figure 6.6
Nanoprogram object code for PU0.

MEMORY MAP

MICRO ADDRESS	NANO ADDRESS			
100	PU0	100	101	
	PU1	100		
	PU2	100		
	PU3	100		
101	PU0	102		
	PU1			
	PU2			
	PU3			
102	PU0	103	104	105
	PU1	103	104	105
	PU2	103	104	105
	PU3	103	104	105
103		TBN	INST	----> TMA=-2
	PU0	106		
	PU1			
	PU2			
	PU3			
104	PU0	107		
	PU1	107		
	PU2	107		
	PU3	107		
105		B	INST	----> MAD=0002

Figure 6.7
Control memory map.

6.3.4 Experimental Results

We performed an experiment in order to evaluate the effectiveness of the microassembly language and its processor on such items as (1) the way in which the two-level microprograms are described by using micro-nano combined statements, (2) the way in which the parallel operations are described by using micro-nano combined statements and parallel nanostatements, and what the degree of parallelism is, (3) the effectiveness of the translation process for supporting high level description, (4) the effectiveness of optimization I, and (5) the effectiveness of optimization II.

Ten problems were given to the members of our laboratory who were knowledgeable about the microassembly language and support system on the console processor. The results are shown in table 6.2. Notice that capital letters in parentheses in the following description correspond to items in the table.

1. Description of two-level microprograms: Microinstructions are described in microstatements (B), which do not include nano level operations, and micro-nano statements (D). The micro-nano statements (D) account for 33.1% of the total microinstructions (B+D) used in the microprograms. This significant usage demonstrates the effectiveness of the micro-nano combined statements for describing tightly coupled micro-nano operations.

2. Description of parallel operations: The description of parallel processing is 27.0% of total descriptions. It may be classified into three categories: (i) micro-nano combined statements (33.3%), (ii) parallel nano statements (41.9%), and

Table 6.2
Experiment results at microassembly language level^a

Microprogram number	1	2	3	4	5	6	7	8	9	10	
Number of statements	(A)	106	138	179	134	144	61	145	187	159	95
Micro- 1 m - 0 n	(B)	35	35	76	38	51	20	35	64	33	23
Nano 0 m - 1 n	(C)	33	79	61	55	32	30	37	78	102	59
0 m - 2 n	(D)	4	12	0	0	6	3	6	0	0	0
0 m - 3 n		0	0	0	0	3	1	5	2	0	0
0 m - 4 n		18	0	4	7	30	0	37	21	6	3
Micro- 1 m - 1 n		9	5	11	19	13	1	6	9	2	2
Nano 1 m - 2 n	(E)	0	4	0	11	0	0	6	1	0	0
1 m - 3 n		0	0	0	0	0	2	1	0	0	0
1 m - 4 n		7	3	27	4	9	4	11	12	16	8
Number of nanoinstructions after decomposition											
PU0	(F)	63	7	93	87	54	35	88	75	53	30
PU1		31	77	34	18	58	7	68	60	56	31
PU2		28	41	34	17	55	7	69	47	43	23
PU3		28	3	35	20	54	10	51	45	40	21
Total		150	128	196	142	221	59	276	227	192	105
Microinstruction utilization	(G)	6	11	54	25	18	7	9	31	30	8
Nanomemory allocation	(H)	0.61	0.41	0.52	0.43	0.83	0.39	0.79	0.80	0.86	0.84

a. m: microinstruction; n: nanoinstruction.

(iii) different nano statements (24.8%). Item (i) corresponds to (D), and items (ii) and (iii) are included in (C). These results show the effectiveness of our approach for describing parallel processing by the micro-nano combined statements and the parallel nanostatements for the SIMD operations. In item (i), the ratio between 1 micro - 2 nano, 1 micro - 3 nano, and 1 micro - 4 nano is 7:1:34 (see (D)). This implies that four PUs are effectively utilized for various problems.

3. Translation of two-level microprograms: The number of nanoinstructions (F) after decomposition is 2.3 times that of explicitly described nanoinstructions (C). This is due to (i) the implicit description in the micro-nano combined statements and (ii) the parallel nanostatements that allow the user to describe parallel operations in one statement. The item (E), the number of nanoinstructions for each PU after decomposition, shows the locality of multiple processor usage.

4. The effect of nanoprogram utilization (optimization I): The numbers of utilized nanoinstructions from multiple microinstructions are indicated by (G). This enhances the nanoprogram memory assignment ratios. However, the effect largely depends on the contents of each microprogram.

5. The effect of nanoaddress compaction (optimization II): The compaction further enhances the nanoprogram memory assignment ratio (H). Microprograms 7 through 10, written by the staff members, attain the ratios around 80%. A distorted use of specific PUs is the reason for the low ratios for microprograms

2, 3, 4, and 6, which were made by student members.

6.4 Optimizing Loader

6.4.1 System Configuration

At the first stage of the machine's development, we did not quite understand the necessity for a microprogram loader. However, the increase in the number of microprogram steps requires a relocatable loader that receives the object microprograms of the absolute assembler and then links and relocates them. It enables the user not only to develop a large microprogram by linking small microprograms but also to perform intermicroprogram module optimization [BABA84].

The loader optimization conditions are basically the same to those for the microassembler, i.e., nanoprogram utilization and compaction conditions. However, they are applied not in intramodule fashion but in intermodule fashion, which means optimization between object microprograms outputted from the microassembler. Thus, the first task of the loader is to analyze the status of the intramicroprogram optimization.

Figure 6.8 depicts the system configuration. It consists of four phases.

- o Phase 1 analyzes the input microprogram modules and finds the state of optimization.
- o Phase 2 performs the utilization of nanoprogram modules among microinstructions in different object modules. The

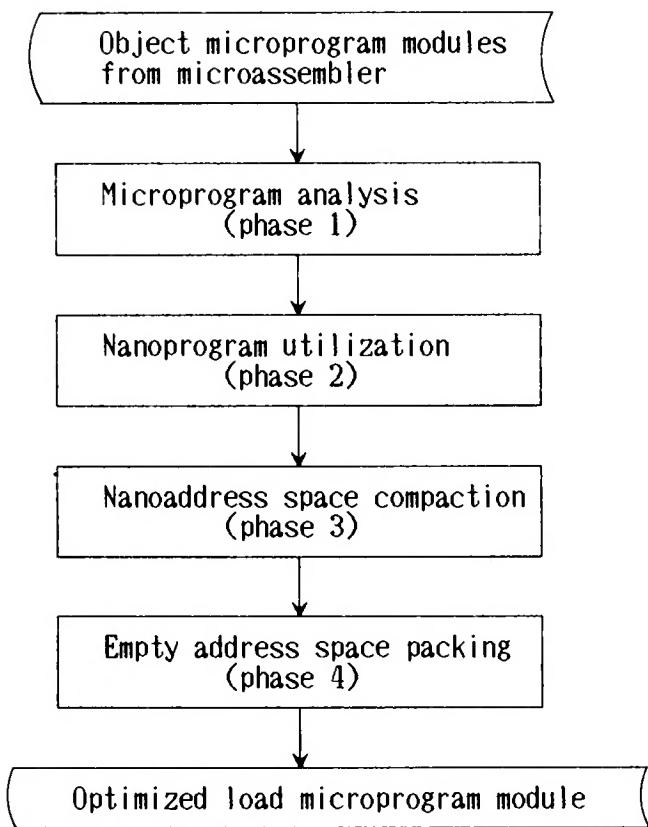


Figure 6.8
Flow of the optimizing loader.

optimization is based on conditions 6.1 and 6.2 of optimization I in the microassembler.

- o Phase 3 compacts the nanoaddress space. The code movement for the compaction is based on condition 6.3 of optimization II in the microassembler.
- o Phase 4 packs empty nanoaddresses yielded by phases 2 and 3.

6.4.2 Experimental Results

The objective of the experiment here is to clarify the effect of

the optimizations for a large scale of microprograms. Therefore, we applied the optimizing loader for five microprograms, designated A to E, written by different users. The results are shown in **table 6.3**. Each microprogram consists of 3 to 10 modules (assembled objects). The numbers of steps are 46 to 490 for microprograms and 52 to 301 for nanoprogams. As the results of the application suggested the relationship between the microprogram size and the effect of optimization, we further applied the loader to all combinations of the microprograms. We summed up the results as follows:

1. The effect of optimization: The effect of the utilization (optimization I) is seen in the number of decrements of the nanoinstructions (NT-ANT). This value changes from one processor to another according to the PU usage patterns. Optimization I reduced the number of nanoaddresses and enhanced the assignment ratio. The effect of the nanoaddress compaction (optimization II) is found as a decrease in the nanoaddress space and an increase in the ratio of nanomemory assignment. Clearly, compaction in both directions gives the best results. The assignment ratios are about 70-90%.

2. Microprogram size and effect of compaction: **Figures 6.9 and 6.10** show the relationship between the size of microprograms and the effect of optimizations I and II. These results demonstrate that there is a strong positive correlation (the correlation coefficients are 0.94 for figure 6.9 and 0.92 for figure 6.10).

Table 6.3
Results of loader optimization^a

Microprogram	A	B	C	D	E
Number of modules	3	8	6	10	8
Microprogram size	48	195	46	168	490
Number of nanoinstructions					
Original (NT)	122	208	141	425	858
After optimization I (ANT)	119	193	123	370	777
Deccrements (NT-ANT)	3	15	18	55	81
Number of nanoaddresses					
Original (M)	52	75	54	131	301
After optimization I (M1)	51	69	48	113	279
After forward optimization II	43	54	44	104	254
After backward optimization II	45	58	45	105	267
After both-way optimization II (M2)	43	53	44	100	252
Assignment ratio					
Original (M)	58.6	69.3	65.2	81.1	71.2
After optimization I (M1)	58.3	69.9	64.0	81.8	69.2
After forward optimization II	69.1	89.3	69.8	88.9	76.4
After backward optimization II	66.1	83.1	68.3	88.0	72.7
After both-way optimization II (M2)	69.1	91.0	69.8	92.5	77.0
Nanoaddress decrement ratio					
After optimization I (D1)	1.9	8.0	11.1	13.7	7.3
After Optimization II (D2)	15.4	21.3	7.4	9.9	9.0
Total (D)	17.3	29.8	18.5	23.6	16.3
Maximum assignment ratio					
Before optimizations (MR)	69.3	86.6	71.9	92.3	76.3
After optimizations (AMR)	69.1	91.0	71.5	92.5	77.0
Processing time [sec]	1.0	7.7	2.2	19.5	81.9

a. key:

$$R = (NT/4M)*100; \quad R1 = (ANT/4M)*100; \quad R2 = (ANT/4M')*100;$$

$$D = \{(M2-M)/M\}*100; \quad D1 = \{(M1-M)/M\}*100; \quad D2 = \{(M2-M1)/M\}*100;$$

MR = {NT/(maximum of the number of nanoinstructions over 4 PUs
before optimization I * 4)}*100

AMR = {ANT/(maximum of the number of nanoinstructions over 4 PUs
after optimization I * 4)}*100.

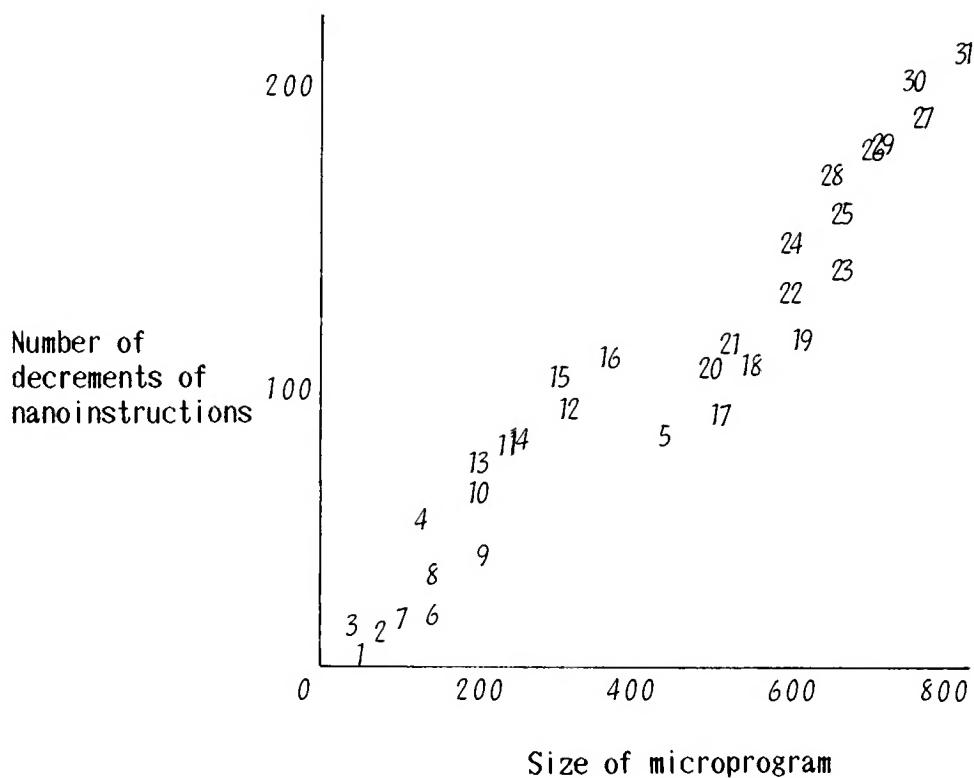


Figure 6.9
Effect of optimization I .

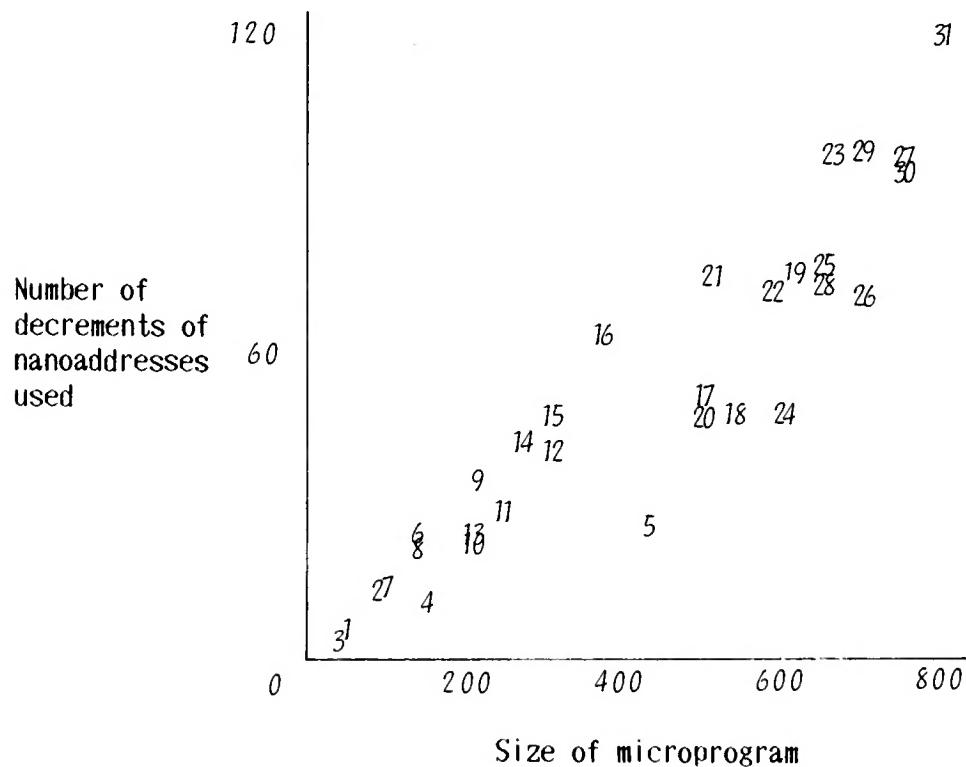


Figure 6.10
Effect of optimization II .

3. Effect of optimization I: The utilization condition, condition 6.1, suggests that we can expect a large effect if there is a repetition of the same nanoprocess. However, in general the program size is the basic factor in determining the effect.

4. Effects of optimization II: To clarify the effect of optimization II, we define the upper limit of the address assignment ratio as follows:

$$\text{maximum assignment ratio (AMR)} = \frac{\text{total nanoinstruction number (ANT)}}{\text{(the maximum nanoinstruction number among 4 PUs) } \times 4}.$$

Comparison of the ratio AMR with the actual ratio M2 in table 6.3 shows that the performance is almost 100% for the microprograms A to D. Furthermore, the comparison for all the combinations of the microprograms showed 96% performance except for the combinations AC, BC, and ABC (microprogram numbers 7, 8, and 9, respectively). Analysis of the microprograms shows that (i) if we try to utilize the multiprocessors as evenly as possible, we can attain a good ratio (for example, if we use PU#0,1 at one place, then PU#2,3 should be exclusively used at another place), and (ii) the PU usage is mainly controlled by the careful assignment of variables to processor facilities before writing a microprogram.

5. The decrement of nanoaddresses by optimizations I and II: D1 and D2 in table 6.3 are the decrement ratios of nanoaddresses by optimizations I and II, respectively. The results show that

the effect varies depending on the contents of the microprograms. For example, optimization I had a larger effect for microprograms A and B, but not for C, D and E. The total reductions (D in table 6.3) are from 17.3 to 31.0%.

These results suggest that intermodule optimization, applied to output modules from the microassembler, enhances the compactness of nanoprogams by utilization and address compaction techniques.

6.5 Debugger

To verify the correctness of microprograms, the first type of support software is *simulators*, which perform the operations of microprograms. They are generally implemented by conventional software programs. Due to the complexity of microinstructions and the parallelism between so many microoperations, such implementations may be very slow. Sometimes it is very difficult to simulate parallel operations by using sequential programs.

When the microprogrammed computer has been built, the second type of support system, called *debugger*, is possible. Its role is to let the computer execute the microcode directly while maintaining a user interface to control microprogram execution. Debuggers are often used to check for timing and machine environment problems that may be difficult to find in software simulations.

Concerning the complexity of the MUNAP microarchitecture, we had decided to develop a microprogram debugger. The debugger

allows the user to load and run the microprograms and monitor their execution. As the debugger of a two-level microprogrammed multiprocessor computer, it activates, stops, and monitors the nanoprocessors as well. **Table 6.4** describes the debugger commands. The debugger was developed mainly on a console processor, ECLIPSE S/130, also utilizing two-level microprograms

Table 6.4
Debugger commands and functions

Function	Contents (COMMAND)
Initialize	Reset MUNAP (GRS); Load debugger micro- and nanoprocessors (IZP)
Start and stop	Start the execution of a microprogram (EXC), or nanoprocessors (EXCN); Stop the execution (STP)
PU specification	Specify PUs to be operated by the other commands such as EXCN and read/write (SPU)
Execution mode	Normal run (SNR), step run (SRN), step run and display (BRK)
Save and restore	Save the contents of microprogram memory, nanoprocessor memory, or main memory (SAV); Restore the saved contents (OLD)
Display status	Display the contents of flags that indicate MUNAP is running or not, interrupt state or not, etc. (HLC)
Read and write	Read the contents of facilities (R<facility name>); Write the contents of facilities (W<facility name>)

and interface hardware of MUNAP. For example, reading and writing of the facilities are basic functions of the debugger. The support hardware is provided to activate a specific microprogram, which in turn reads the contents of the specified facility and sends them to an interface register to be displayed by the console processor or to write the contents of the interface register into the specified facility.

Figure 6.11 shows a procedure for loading and executing the object code of the bit count microprogram, which was translated from the source microprogram, shown in figure 6.1, by the microassembler.

6.6 Evaluator

6.6.1 Design Objectives

For an experimental computer, it is very important to collect data that can be used for evaluating architectural features and thus for directing future improvement. In particular, we need to collect dynamic data, because static data are to be obtained by microassembler. We developed an evaluator for MUNAP that can collect data while running the machine step by step [KATA83]. According to the design principles, the following functions were implemented:

1. **Evaluation of parallelism:** The parallelism may be evaluated by the number of active processor units for each micro cycle. They may be further divided into SIMD and MIMD types.

```

: OLD BITCTO          ;Load object microprogram
                      ;from file BITCTO

: RMPM 100,105        ;Read microprogram memory
100 C02100F           ;to confirm the contents
101 B1C1028
102 008103F
103 E9C1068
104 180107F
105 D600020           ;Read nanoprogram memory
                      ;four columns correspond to
                      ;four PUs

:RNPM 100,107
100 D000004080 D000004081 D000004081 D000004081
101 D000004881 00000000000 00000000000 00000000000
102 4C81914DC1 00000000000 00000000000 00000000000
103 C0018F8400 C0018F8400 C0018F8400 C0018F8400
104 AC100840C0 AC100840C0 AC100840C0 AC100840C0
105 4C8281C091 4C8281C081 4C8281C081 4C8281C081
106 E03DEC0001 00000000000 00000000000 00000000000
107 58828FCDC1 4C829BCDC1 4C829BCDC1 4C829BC101

:WMM 4                ;Set input data which include
= 0123456789ABCDEF   ;133 (85 in hex) 1's
= 5555AAA77778888
= FFFFFFF4443222666
= 010215990FFCBDDD

:EXC 100              ;Execute the bit-count microprogram

:RRF 1,4              ;Read register files in four PUs
1 0000 0000 0000 0000
2 001E 0021 0023 0023
3 0002 0007 000A 000C
4 0000 0000 0000 0085
                      ;the answer (85 hex) is left in RF3(4)

:*

```

Figure 6.11
Load and execute a sample bit count microprogram.

2. Evaluation of two-level microprogramming scheme: The scheme may be evaluated from several viewpoints. The number of nanoprogram steps for each microinstruction gives an idea on the effect of local control in each PU. The data on machine cycles may be divided into three categories: (i) a microinstruction is executed, (ii) nanoinstructions are executed, or (iii) both a

microinstruction and activated nanoinstructions are executed. (Notice that the nanoinstructions were activated by the previous microinstruction; see subsection 3.4.1.) The usage frequencies of micro- and nanoinstructions were expected to give detailed information on the effectiveness of two-level microprogramming.

3. Evaluation of nonnumeric functions: The usage frequencies of micro- and nanoinstructions also show the effectiveness of the nonnumeric functions, because vertically structured instructions are in one-to-one correspondence with the functions. Furthermore, the usage frequencies of each microorder gives more specific information on the effectiveness.

6.6.2 Data Collection and an Example

According to the above functional requirements, the evaluator was designed to collect the following data: (Notice that the output lists from the evaluator are all for the bit count microprogram shown in figure 6.1.)

1. Microlevel data: Figure 6.12a shows example data for the microinstructions from AA to MPMW that can activate nanoprogams. The last NN accounts for special machine cycles where there is no corresponding microinstruction because nanoprogams are running by themselves and the microinstruction that activated them is not running at the microcycle. Active processor units in each machine cycle are indicated by ones in a 4-bit string at the leftmost part of each column. For example, 0011 represents that PUs 2 and 3 are active. S and M correspond to SIMD and MIMD,

	AA	AB	BA	CB	AM	BN	TBN	MPMW	NN
0000	0	0	0	0	0	0	0	0	0
0001	0	0	0	0	0	0	0	0	0
0010	0	0	0	0	0	0	0	0	0
0100	0	0	0	0	0	0	0	0	0
1000	0	0	0	0	4	0	4	0	1
0011 S	0	0	0	0	0	0	0	0	0
0011 M	0	0	0	0	0	0	0	0	0
0101 S	0	0	0	0	0	0	0	0	0
0101 M	0	0	0	0	0	0	0	0	0
1001 S	0	0	0	0	0	0	0	0	0
1001 M	0	0	0	0	0	0	0	0	0
0110 S	0	0	0	0	0	0	0	0	0
0110 M	0	0	0	0	0	0	0	0	0
1010 S	0	0	0	0	0	0	0	0	0
1010 M	0	0	0	0	0	0	0	0	0
1100 S	0	0	0	0	0	0	0	0	0
1100 M	0	0	0	0	0	0	0	0	0
0111 S	0	0	0	0	0	0	0	0	0
0111 M	0	0	0	0	0	0	0	0	0
1101 S	0	0	0	0	0	0	0	0	0
1101 M	0	0	0	0	0	0	0	0	0
1011 S	0	0	0	0	0	0	0	0	0
1011 M	0	0	0	0	0	0	0	0	0
1110 S	0	0	0	0	0	0	0	0	0
1110 M	0	0	0	0	0	0	0	0	0
1111 S	4	0	0	0	0	1	0	0	4
1111 M	1	0	0	0	0	0	0	0	4
TOTAL	5	0	0	0	4	1	4	0	9

(a)

	LT	B	TB	MNFC
TOTAL	0	1	0	0

(b)

Figure 6.12
Micro usage frequencies for the bit count microprogram:
 (a) nano activating microinstructions; (b) nano independent microinstructions .

respectively. Thus, '0011 S' means that PUs 2 and 3 do the same operation. **Figure 6.12b** shows usage frequencies for the microinstructions that do not activate nanoprograms.

Using the data in figure 6.12, the average number of activated processor units is computed as follows:

$$\frac{\sum (\text{the number of activated PUs}) \times (\text{microsteps})}{\sum (\text{total machine cycles})}, \quad (6.1)$$

where the number of activated PUs is the number of 1's in the bit string, and the the number of microsteps is indicated as the numbers in the corresponding row; the total number of machine cycles can be computed in two way: one is to add up all the microsteps in the table, and the other is to exclude the microsteps where no PUs are activated.

The two-way computations yield an average number of activated PUs (a) for all the microinstructions executed and (b) for nano activating microinstructions executed.

2. **Nanolevel data:** Figure 6.13 shows an example output for nanolevel data. Each row corresponds to a nanoinstruction, and contains the usage frequencies of the nanoinstruction for PUs 0, 1, 2, and 3. Figure 6.14 is the nanostep table, which counts the number of nanoprogam steps for PUs 0, 1, 2, and 3. For example, the number 4 at the crossing of the fourth row and the column for PU#0 indicates that PU#0 executed 4 three-microstep nanoprogams.

Using the data in figure 6.14, we can compute the average number of nanoprogam steps as follows:

$$\frac{\sum (\text{the number of nano step}) \times (\text{the number of nanoprogams})}{\sum (\text{the total machine cycles})}. \quad (6.2)$$

The total number of machine cycles can be computed in two ways, as indicated by equation (6.1). MNPSTEP ALL and MNPSTEP represent the average numbers for all the microinstructions

NANO TABLE

	PU#0	PU#1	PU#2	PU#3	TOTAL
BOU	4	4	4	4	16
ALU	9	5	5	5	24
DCU	0	0	0	0	0
EX	4	4	4	4	16
NTB	4	0	0	0	4
NLT	2	1	1	1	5
NOP	0	0	0	0	0

Figure 6.13
Nano usage frequencies for the bit count microprogram.

NANO·STEP

	PU#0	PU#1	PU#2	PU#3	TOTAL
0	0	8	8	8	24
1	9	2	2	2	15
2	1	0	0	0	1
3	4	4	4	4	14

MNPSTEP	1.16
MNPSTEP ALL	1.12

Figure 6.14
Nanoprogram steps for the bit count microprogram.

executed and for nano activating microinstructions, respectively.

3. Accumulated data: Utilizing the data at the micro- and nanolevels, we can compute the following data: (1) *The number of machine cycles:* The total machine cycles may be decomposed into three categories, namely, microcontrolled cycles, nanocontrolled cycles, and micronano combined cycles (see subsection 3.4.1). In the case of the sample bit count microprogram, the total number of machine cycles is 25. According to the above criteria, it is decomposed into 2, 9, and 14, respectively. (2) *Execution time:* Usual microinstructions have the same execution time of 550 nanoseconds. However, the microinstructions, which access MM or do a serial operation, require a longer execution time (extra

100 nanoseconds). Thus, the microprogram execution time may be computed concerning these differences. In the bit count microprogram there are 8 such microinstructions and the execution time was computed as 14.5 microseconds. (3) *Parallel PUs*: By collecting the number of active PUs for each machine cycle, we can compute the average number of active PUs as follows:

$$\frac{\sum (\text{the number of active PUs}) \times (\text{the number of machine cycles})}{\sum (\text{total machine cycles})} \quad (6.3)$$

Again, the total number of machine cycles can be computed in two ways. We named the average numbers of active PUs "PARA PU ALL" and "PARA PU" for all the microinstructions and for nano activating microinstructions, respectively. In the case of the bit count microprogram, the evaluator counted nine 1-PU cycles and fourteen 4-PU cycles, and computed PARA PU and PARA PU ALL to be 2.82 and 2.60, respectively.

6.6.3 Other Optional Data

The other data that may be collected by the evaluator include the count of execution frequencies of instructions at specific micro- and nanoaddresses or in a specific range of addresses.

Furthermore, the later experiment for architectural evaluation and improvement required other information not included here. For example, usage patterns of series of micro- or nanoinstructions were required for the research on architecture synthesis. These optional functions have been added to the original evaluator.

Applications

7.1 Basic Concepts for Application

This chapter describes the application of MUNAP to various areas, such as emulation, programming language processing, database systems, graphics, and numerical computation. Our objective for applying the basic hardware and support software system to these areas is to clarify the effect of the architectural features in real environments. For example, in a given application environment, how many processors are utilized effectively; how do the nonnumeric oriented functions work; and what is the control scheme of two-level microprogrammed multiprocessor architecture? It is likely that the effect varies from one application to another. We believe that the application of a real machine to a real environment will be the best way to reach correct answers to these crucial issues. Many performance studies have been done

using paper machine models. However, the problem is that the assumptions and inadequacies of such models sometimes miss important issues, which cannot be overlooked if we use a real machine. For example, usually the parallelism of a real machine cannot be fully utilized; the pre- and postprocessing for parallel operations require extra operations, and we should be utilizing a limited parallelism of the real machine to solve a problem with large parallelism.

According to our objective, several topics were selected. Needless to say, the first consideration was given to promising areas where the architectural features seem to fit the needs. In particular, we sought out areas where efforts to overcome the drawbacks of a conventional von Neumann architecture have been exerted. These include the revolution in memory structures, such as tagged architecture and abstract data type, and the emergence of new programming paradigms, such as logic programming and object oriented programming. We also selected areas with explicit parallelism. The reason for this selection is to test the limitation of parallelism by four PUs as well as the applicability of so-called nonnumeric functions to numerical problems. All of these applications were aimed at clarifying the effect of MUNAP's architectural features.

Consequently, the following projects were underway, as indicated in the introduction by the spring of 1986:

1. emulation of a minicomputer,
2. tagged architecture for a system description language,

3. architecture support for software testing,
4. firmware implementation of abstract data types,
5. Prolog high level language machine,
6. Smalltalk-80 high level language machine,
7. color graphics system,
8. numerical computation for Fast Fourier Transform (FFT) and LU decomposition for simultaneous linear equations.

The first emulation project is a basic application of a microprogrammable machine. The second and third tagged architecture projects are our effort to challenge the drawbacks of von Neumann machines by adding meaning to data on memory. The fourth abstract data type project implements the concept through firmware for relational database processing. The fifth and sixth projects are to construct high level language machines for Prolog and Smalltalk-80, which attracted our attention as relatively new languages for artificial intelligence. The seventh and eighth projects differ from the others because the areas include explicit parallelism so as to process large amounts of homogeneous data that seem to be beyond the parallelism of four processor units. As will be seen later, the seventh project has another important objective as an application of the language developed for the third project.

In the following sections, we shall introduce each project by describing (a) the motivation for each project, (b) the outline of the processing, and (c) the results of the application. Throughout the descriptions we shall try to clarify the effect of architectural features for each application. The

total evaluation will be done in the following chapter.

7.2 Emulation of a Minicomputer

7.2.1 Definition and Background

Emulation is defined here as a combined hardware-firmware approach to the process of representing the functioning of one machine instruction set by another. The term *emulator* is used to describe a complete set of microprograms, that, when embedded in a control store, define a machine [HUSS70, AGRA76]. The machine that is realized by an emulator is called a *target machine*, and the machine that supports microprograms is called a *host machine*. Historically, it has been an important approach to converting a program written for a machine so as to run on another machine. To define a machine, the microprograms of the host machine should emulate a conceptual structure and a functional behavior of the target machine by (a) mapping the components of the target machine into those of the host machine and (b) performing the machine language instructions of the virtual machine.

7.2.2 Emulation of a Minicomputer

We selected the ECLIPSE S/130 minicomputer [DATA77] as a target machine of our emulation because we have the machine in our laboratory. Table 7.1 shows a comparison of the target and host machines [SASA85]. The ECLIPSE machine instructions may be divided into those for usual functions and the others for

Table 7.1
Comparison between target and host machines

	Target (ECLIPSE)	Host (MUNAP)
Control	56-bit horizontal microinstruction	Two-level microprogramming
Machine cycle time	200 nsec	550 nsec
Processing unit	16 bits x 1 PU	16 bits x 4 PU
Special hardware	Floating point data exchanging	Field/bit operations
Registers	4 accumulators	16 registers x 4 1 K scratchpad memory x 4
MM addressing	Bit, byte, word	1, 2, 4, 8 bytes

floating point operations. The instruction lengths may be one or two words. Units of addressing may be a word, a byte, or a bit. Basically, the effective address is computed using a displacement and the contents of an accumulator.

The resources of ECLIPSE were mapped mainly onto the register file (RF) and scratchpad memory (SPM) of MUNAP. PU#0 was used as the main processor. Instruction registers were replicated among four RFs so that they can be decoded in parallel. Memory address register, program counter, and floating point status register were allocated to PU1-3 so that they may be handled in parallel with the main operation in PU0. A 64-bit floating point accumulator is allocated to SPM0-3 with the same address so that

data in the SPM may be processed in parallel. There are several work areas that do not have direct correspondence with the resources of ECLIPSE. They are used to store temporary results during computations.

The execution is divided into three phases: fetch, decode, and execution. In the fetch phase, a 64-bit word is accessed at a time. It saves an access time for 2- or 3-word instructions. In the decode phase, the instruction with complex formats may be decoded in parallel in four PUs. The execution phase consists of microroutines of the following four categories:

1. Microroutines for ordinary instructions, such as fixed point arithmetics, logical operations, and bit and byte operations. These functions were implemented by nanolevel operations, such as ALU, BOU, and DCU.
2. Microroutines for floating point instructions. As the floating data are 64-bit, the four PU operations were effectively utilized. Several generic routines, such as normalization and sign and exponent parts extraction, were developed.
3. Common microroutines used for (1) and (2). They include effective address calculation, carry handling, and condition test. The condition test is done in parallel in four PUs.
4. Microroutines for system call. The system call was mapped onto a call from MUNAP to the service processor, ECLIPSE, to ask some service. (Notice that ECLIPSE has two meanings for this project: a target machine of emulation and a service processor

for MUNAP.)

7.2.3 Evaluation

The amounts of microprogram and nanoprogram were 1.7 K and 1.1 K x 4 words, respectively. We defined 9 problems, 6 for usual (group A) and 3 for floating point (group B). The results show that the decoding and generic routines occupy high percentages. The ratios of MUNAP execution steps to that of ECLIPSE were about 5 for group A and 8 for group B. The average PU utilization is 2.1 for group A and 2.4 for group B. The relatively high ratios are based on the complex instruction formats of ECLIPSE, the sequential nature of the processing in the unit of 16 bits, and the special hardware for floating point operations in ECLIPSE. For the reduction of the ratios, we need to speed up the execution time of decode and floating point operations, and make the call/return overhead of microroutines much smaller (now it occupies about 6% of the execution time).

7.3 Tagged Architecture for a System Description Language

7.3.1 Design Principle and Language Specification

System description languages have been designed as an answer to two contradictory demands: ability improvement of description language and avoidance of the low efficiency of target program execution. Several proposals have been made to find a proper compromise between them. The PL 360 [WIRT78] and C [KERN78] are

the typical examples designed for IBM System 360 based on PL/I and for the UNIX operating system, respectively.

After developing a microassembly language, we felt that it was still difficult to describe large utility programs or application programs in such a language. Thus, we decided to develop the MUNAP System Description Language (MSDL) with the following objectives [YAMA84a,b, BABA83a]:

1. **Definition of high level architecture:** On the microassembly language architecture, a rich set of data types, operators, and functions should be defined. The examples include data types of two-dimensional array and structure, control statements of IF, WHILE, FOR, and SWITCH, operators for data exchange and concatenation, and functions for bit and string operations.
2. **Utilization of hardware features:** To utilize the hardware features of MUNAP, we represent some of them in the language. Examples are string functions for nonnumeric units, such as BOU and DCU, and shift and exchange operators for the SEN. Some of the data types, such as flag, are also represented. This is a compromise between the high level, problem oriented architecture and low level hardware organizations.
3. **Tagged architecture:** The goals of an effective computer architecture are not only the efficient processing of large amounts of data but also the enhancement of reliability of the debugging process and the computing system [MYER79]. Higher processing capability should be applied not only for processing

large amounts of data at high speed but also for improving the user interface by semantic checking during program execution. To implement these concepts at the system description level, we designed the tagged architecture. This architecture is expected to provide the facilities for (i) detecting several kinds of errors at run time, such as referring to an unassigned data value, and (ii) automatically transforming the data types of operands. These two items aid the development process of programs.

Table 7.2 shows the data types of MSDL. The basic unit of data lengths is 16 bits, which corresponds to a bit length of PU. The data types of flag and operator correspond to hardware flags and functional control of operation units, respectively. Two-dimensional arrays and structures are also provided that are to be efficiently supported by an AM-MM combined structure.

Table 7.2
Data types

Type	Declaration
Integer	SHORT (16) ^a , INTEGER (32), LONG (64)
Real	FLOAT (32), DOUBLE (64)
Bit	SBOOL (16), BOOL (32), LBOOL (64) LBOOL n (64+16n)
Character	CHAR n (8n)
Flag	FLAG (1)
Operator	OPERATOR (8)

a. The numbers in parentheses denote bit lengths.

Table 7.3 shows operators. Arithmetic and logic operations correspond to the ALUs, and shift and exchange operations to the SEN functions. **Table 7.4** shows the string functions, which may be frequently used in system programs, but cannot be supported efficiently by a conventional computer.

Figure 7.1 shows an example of MSDL description. In this example, the definite integral of $S = \int_0^2 (x^3 + 2x - 1) dx$ is evaluated according to Simpson's formula with the division of 10. The description is simplified through the control statements like C and through the one-statement description of the variable declaration and initialization.

7.3.2 Parallel Processing of MUNAP System Description Language
 Basically, the source program written in MSDL is translated into an intermediate form by ACOS600S (see figure 5.3). The intermediate language programs are transferred to the service processor ECLIPSE and then interpretively executed by MUNAP microprograms. In order to make use of the MUNAP microarchitecture features, the intermediate language has one-to-one correspondence with the MSDL source statement. The translator and the interpreter are described in the ACOS-PASCAL and the MUNAP microassembly language, respectively. The interpreter consists of 3.2 K microinstructions and 7.6 K (1.9 K for each PU) nanoinstructions.

We shall concentrate on the processing features supported by the parallelism and nonnumeric functions of MUNAP.

Table 7.3
Operators

Type	Operator
Arithmetic	*(multiply), /(divide), %(remainder), +(add), -(subtract)
Logic	!(not), &(and), (or), ^(exclusive or), !&(not and), ! (not or), !^!(not exclusive or)
Shift	SLL, SRL, SLA, SRA, SLC, SRC (up to 63-bit shift)
Exchange	M8, M16, M32, M64 (mirror conversion)
Relation	XX, ** (BADC, DCBA data exchanges, see Fig. 3.3)
Combination	>, <, >=, <=, ==, != &&,

Table 7.4
String functions

	Description	Operation
Bit string	BSUBSTR (BIT, POS, N)	Extraction of N bits from bit position POS
	PEC (BIT, L/M. 0/1)	Detection of 0/1 from LSB/MSB
	BCT (BIT, 0/1)	Counting of 0/1
	BCON (B1,B2, ..., BN)	Concatenation of bit strings B1, B2, ..., BN
Character string	CSUBSTR (CHAR, POS, N)	Extraction of N characters from character position POS
	INDEX (CHAR, L/M, C)	Finding of character C from LSD/MSD
	CCT (CHAR, C)	Counting of character C in CHAR
	CCON (C1, C2, ..., CN)	Concatenation of character strings C1, C2, ..., CN

```

PROCEDURE INTEGRAL;
LOCAL INTEGER SPM I,M=20;
    FLOAT   SPM (S,X)=0.0,F,H,
            A[3]=-1.0,2.0,0.0,1.0;

PROCEDURE FX(X,F);
    FLOAT X;
    VAR FLOAT F;
    LOCAL DOUBLE   SPM Y;
    INTEGER   SPM I;

{ Y=0.0;
FOR ( I=3; I>=1; I=I-1)
{ Y=(Y+A[I])*X; }
F=Y+A[0];
}

{ H=2.0/M;
FOR ( I=2; I<=M; I=I+2)
{ FX(X,F); S=S+F; X=X+H;
  FX(X,F); S=S+4.0*F; X=X+H;
  FX(X,F); S=S+F;
}
S=H*S/3.0;
}

```

Figure 7.1
Integral computation program in MSDL.

The use of macroinstructions: The parallelism coupled with a two-level microprogramming scheme makes the microprogramming process much more difficult than that for a conventional machine. Our experience taught us that the microprogram, about 50 steps, can be made and maintained fairly easily. Therefore, we defined macroinstructions for the basic functions commonly used among different modules of the interpreter. They include transfers between the MM and SPM, floating point number processing, saving and restoring the contents of RF, and so on. They were expected

to make the development process simpler and decrease the size of the whole interpreter.

Parallel processing for operators: The arithmetic and logic operations are executed in parallel in the four PUs for each operator of MSDL. The type check for operands and, if necessary, the translation to an appropriate type are also done dynamically. The information on the result of the operation is stored in a tag field, to be described later.

As an example of parallel processing, we show the outline of the shift operator microroutine that does an N-bit circular shift on the 64-bit data. As 4 bits are the smallest unit of the SEN operation, the SEN shifts the data $4 \times D_4$ ($D_4 = N \text{ div } 4$) bits in one machine cycle. Then, the ALU shifts it M_4 ($M_4 = N \text{ mod } 4$) bits, one bit by one bit. The use of the SEN reduces the number of microcycles from 31.5 to 2.5 on the average. This example shows not only the enhancement of processing by parallelism but also the provision of a uniform function (in this case shift) to the user. If we use the function at the microassembly language level, we must directly control the SEN and the ALU shifter to get appropriate results.

Parallel processing of string functions: The string functions are executed by using the bit count and priority encoding functions of the bit operation unit (BOU), the field extraction and embedding functions of the divide and concatenate unit (DCU) in the four PUs, and the shift and broadcast functions of the SEN. For example, a bit string extraction function BSUBSTR(BIT,

POS, N) extracts N-bit data from the bit position of POS-bit of 64-bit variable BIT. To do this operation, this routine gets the data from eight banks of MM to register file (RF) in four PUs in parallel. Then the data are left shifted (POS - 1) bits by using the SEN and the ALU shifter. After computing i (= N div 16) and j (= N mod 16), the data are concatenated with 0 at the (j+1)-bit at PU_i and are stored into eight banks of MM in parallel.

This example shows the difficulty and tediousness of handling the multiple processing units, especially if they have certain specific features, such as nonnumeric processing capability. These functions provide the user with a high level but efficient interface by doing tedious and, sometimes, tricky operations for utilizing the parallelism of the microarchitectures instead of the user.

Implementation of tagged architecture: Each variable in MSDL has a 26-bit tag field, as shown in figure 7.2. The check points are divided into two major parts: (a) checks at the fetch and operand access phase and (b) checks at the execution phase. The checks for item (a) include (i) system variable error (check the range of system variables, such as intermediate language instruction counter), (ii) parameter error (check the number, attribute, and order of formal parameters for procedure call instruction), and (iii) access error (check if the MM and SPM addresses point to the user variable area, the operand value is defined, and the index of array is within the correct range). The checks for item (b) depend on the content of processing. In

	4	8	2	2	10	(bit)
Attribute	Capacity	Overflow/ Underflow	Define/ Refer	Number of references		

Figure 7.2
Tagged data structure.

the arithmetic operations, for example, overflow, underflow, and division by 0 are checked. In both (a) and (b), the BOU and DCU functions ease the reference to the tag, which consists of several fields.

These checks may seem to be redundant. However, they not only enhance the user interface but also check erroneous actions caused by incorrect input data. Further, the tagged architecture is made feasible by fast parallel processing of multiple processors.

7.3.3 Evaluation of Architecture Hierarchies and Firmware Interpreter

In order to make the architectural difference clear, the characteristics of the three levels, namely, bare hardware, microassembly language, and MSDL, are summed up in table 7.5. Needless to say, the higher the level of language, the richer the facility. This can be generalized to all the aspects of language, such as the data structure, arithmetic and logic operators, and control functions. As to the description of parallel operations under the two-level microprogramming scheme, MSDL hides the hardware features from the user and problem

Table 7.5
Comparison among three microarchitecture levels

	Bare hardware	Microassembly language	MSDL
LANGUAGE FEATURES			
Data structure	Integer(16),Character, Boolean	Integer(16),Character, Boolean	Integer(16,32,64), Real(32, 64), Character, Boolean
Arithmetic operations	Functions of hardware units, such as ALU	Facilities for micro-nano combined operations	Problem oriented operators and functions
Control	Functions of sequencer	IF, GOTO, CASE that correspond to sequencer functions	IF, WHILE, FOR, SWITCH in problem oriented format
Extensibility	Equals that of hardware	New microinstruction, microinstruction field, module microorder	New microprogram
ARCHITECTURAL FEATURES			
Parallelism	Direct description by users	Single statement for SIMD operations	User independent feature
Two-level microprograms	Direct description by users	Single statement for tightly coupled micro-nano operations	User independent feature
Uniformity	Limited by avoidance of hardware redundancy	Uniformity for some test operations	Uniformity for all the functions
Facilities for program test	Debugger to run and monitor microprograms	Debugger to run and monitor microprograms	Tagged architecture

oriented functions are provided. At the microassembly language level, frequently used micro-nano combinations, and the instructions with the SIMD feature, may be described in one statement. The error check function is only provided at the MSDL level to aid the programming process and enhance system reliability.

In order to evaluate the effectiveness of the MSDL and its processor, we obtained dynamic data in evaluating the example MSDL program shown in figure 7.1. The execution took 0.44 seconds for 725,389 microsteps. Table 7.6 shows the result:

1. The percentage of the fundamental cycle is 24.7%. In the ratio, fetch and decoding occupy a fairly low percentage (3.3%) and the data access and the conversion of intermediate codes are 10.6 and 9.8%, respectively.
2. The steps for the MSDL operators are divided into two parts, i.e., type conversion (6.2%) and arithmetic operation (7.2%).
3. The high usage ratio (46.5%) of macroinstruction routines proves the effectiveness of such utility microroutines.
4. The average number of dynamically active PUs is 2.2.
5. The overhead caused by error checks and related operations is classified into 4 categories as shown in table 7.7. Type check and type transformation are major parts.

Table 7.6
Dynamic evaluation data

Contents	Ratio (%)
Fundamental cycle (Main, Initialize, Fetch, Decode)	24.7
Type conversion	6.2
Arithmetic operation	7.2
Expression	12.1
Statement	1.7
Macroinstruction routines	46.5
Others	1.6

Table 7.7
Processing time ratio for tag operations

	Type check	Type transform	Tag create	Result set
Integer	28.2	46.6	14.5	10.7
Real	31.2	40.7	16.3	11.4

7.3.4 Evaluation of Tagged Architecture

Generally, these are the advantages of a tagged architecture [YAMA84b].

(A-1) While a conventional machine language provides several operation codes for a single operation with several data types, such as add, add double, add floating, etc., a unified operator can support different data types. This reduces the necessary amount of object codes on the MM and microprograms on the control memory.

(A-2) The number of symbol table references decreases because the tag contains the necessary information. Furthermore, the tagged architecture enables type transformation by firmware, while it is done by machine languages in a conventional machine.

(A-3) The one-to-one correspondence between operators and the intermediate language makes the type check of operands unnecessary and thus makes the translation process simple.

(A-4) The tag enables efficient error detection at run time and helps the debugging process.

On the other hand, these were the expected disadvantages:

(D-1) The MM amount increases by the attachment of tag area to each data.

(D-2) The execution steps increase by tag handling and data type

conversion.

There are two trade-offs: one between (A-1) and (D-1), which affects memory amounts of main memory and control memory, and the other between (A-2) and (D-2), which affects execution steps. (A-3) and (A-4) are advantages of tagged architecture.

In order to evaluate the tagged architecture for MSDL from the above viewpoints, we described 10 programs in MSDL. They include 3 floating point arithmetics, 2 character string processings, 3 bit string processings, and 2 others. The results are summed up as follows:

1. **The amount of main memory:** The amount is a trade-off between the tag area (necessary for a tagged architecture) and the data type transformation instruction area (necessary for a conventional architecture). The results show that the trade-off point changes, depending on the features of programs and the tagged architecture, are not necessarily worse than the conventional ones.
2. **The amount of control memory:** Unlike conventional architectures, the tagged architecture can utilize one routine for various lengths of data within each data type. This reduces the amount of microprograms for type transformation and arithmetic operation to 60% of those for a conventional one.
3. **Execution time:** The time is a trade-off between the time for dynamic type check and tag manipulation (necessary for a tagged

architecture) and the time for fetching and decoding type transformation instructions (necessary for a conventional architecture). The result shows that the execution times of both architectures are almost the same. This means that the tagged architecture is more effective because it consumes time for extra operations, such as detection of undefined variables and updating reference count fields.

4. Error detections: 80% of system errors and 50% of program errors were detected by using the tags. They include operand access error and variable reference error.

Furthermore, program improvement is possible by utilizing the reference count field of tags. In order to speed up the tagged architecture, the type transformation routines were singled out as the key to the improvement as they occupy half the execution time for arithmetic operations.

7.4 Architectural Support for Software Testing

7.4.1 Software Testing and Computer Architecture

As software becomes large and complex, the test has very important implications for the enhancement of its quality [MYER79, JOHN82]. This is evidenced by the fact that the cost for software testing is said to be 50% of the total cost for software development. In the field of software engineering, many attempts

have been made, such as methods for static and dynamic analysis of programs to enhance software quality. Interactive debugging systems have been developed to allow the user to control program executions and get evaluation data. However, the overhead incurred by such additional processing makes the execution speed very slow. For example, an experimental result indicates the overhead is 30 times as much as for usual processing. Thus, architectural support is desirable in order to make the approach realistic [CHU82]. The proposals of special hardware mechanisms, such as range checks [HILL81] and detection of undefined references [GERR80], are examples of such support. Several solutions are proposed for overcoming the drawbacks of von Neumann architecture where the instruction defines the attributes of the operands. Tagged architecture is a well-known approach to self-defining data that enables several semantic checks.

Thus, we developed a software testing system on MUNAP utilizing its architectural features [BABA86]. In order to make data self-defining, the tagged architecture and data descriptor are employed. Tagged architecture was described in the previous section. The data descriptor is an extension of the tag that includes less primitive and multidimensional data objects, such as structures and arrays. The descriptors differ from tags in that they are disjoint from the data; they point to the data, and instructions indirectly address the data through the descriptors.

7.4.2 System Configuration

A low level linked list language, L⁶ (pronounced "el six"), was chosen as the target language for a software testing system [KNOW66]. The language has been around a long time, but is not popular. The reason for our choice is twofold: (a) the level is so low that the pure processor for the language is simple and we can concentrate on the support functions, and (b) the rich set of data types and the frequent use of pointers were expected to make the debugging process complicated. Figures 7.3a and 7.3b illustrate an L⁶ program and the data structure made by the execution of the statements until line 16, respectively. In figure 7.3b, each box is called a block and is divided into several fields. A circle, which indicates a start of a reference chain, is called a bug. To refer to a field, the reference chain should be specified by starting from a bug name. We call a line of L⁶ a program statement and a unit of operation, in parentheses, an instruction. For the details of the language, see [KNOW66].

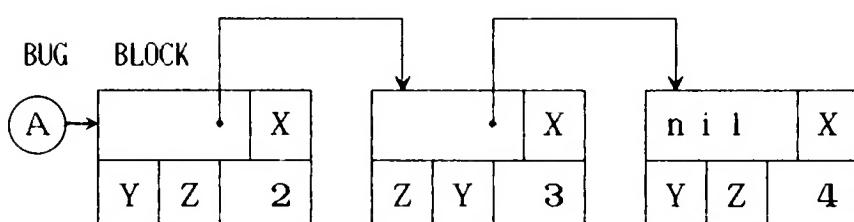
Figure 7.4 shows the system configuration of LIDS (L⁶ Interactive Debug Support system). The user can utilize the support functions interactively. The L⁶ programs are translated into an intermediate language by the translator on ECLIPSE, and then the intermediate language programs are executed in MUNAP by the L⁶ interpreter coupled with the buddy system for free storage allocation. The command interpreter accepts user commands and activates debug support program on ECLIPSE, which further activates debug support microprograms.

```

1 PROCEDURE MAIN;
2   THEN (A,SS,2,Z);           SETUP STORAGE
3   THEN (O,DP,1,3)(0,DX,25,1); FIELD DEFINITIONS
4   THEN (1,DY,1,1)(1,DZ,9,1)(1,DN,17,2);
5   THEN (X,FA,58)(Y,FA,59)(Z,FA,5A)(N,F,4); INITIALIZE
6   THEN (E,GT,1)(E,FR,0);     E <--- NIL
7   THEN (A,P,E);            A <--- E
8   THEN (DO,HANOI);         CALL HANOI
9 END;
10 PROCEDURE HANOI;
11 L10: IF (N,ED,1) THEN (DO,PRINT) L20; IF N=1 THEN PRINT & GOTO L20
12   THEN (A,GT,2,AP);        GET BLOCK (AP<---A)
13   THEN (AX,F,X)(AY,F,Y)(AZ,F,Z)(AN,F,N); AX,AY,AZ,AN <--- X,Y,Z,N
14   THEN (Y,IC,Z);          Y <--- Z
15   THEN (N,SD,1) L10;      N <--- N-1 & GOTO L10
16 L20: IF (E,P,A) DONE;    IF POINTER=NIL THEN RETURN
17   THEN (X,F,AX)(Y,F,AY)(Z,F,AZ)(N,F,AN); X,Y,Z,N <--- AX,AY,AZ,AN
18   THEN (A,FR,AP);         FREE BLOCK (A<---AP)
19   THEN (DO,PRINT);        CALL PRINT
20   THEN (N,SD,1);          N <--- N-1
21   THEN (X,IC,Y) L10;     X <--- Y & GOTO L10
22 END;
23 PROCEDURE PRINT;
24   THEN (2,PRA,0A0D)(6,PRA,204D4F564520); PRINT "<LFCR> MOVE "
25   THEN (D,BD,N)(D,ZB,D)(2,PR,D);        PRINT [N] BY ASCII
26   THEN (6,PRA,2046524F4D20)(1,PR,X);    PRINT " FROM "[X]
27   THEN (4,PRA,20544F20)(1,PR,Z) DONE;    PRINT " TO "[Z] & RETURN
28 END;

```

(a)



(b)

Figure 7.3

Example of tower of HANOI program in L⁶:

(a) example source program; (b) example blocks (status at line 16).

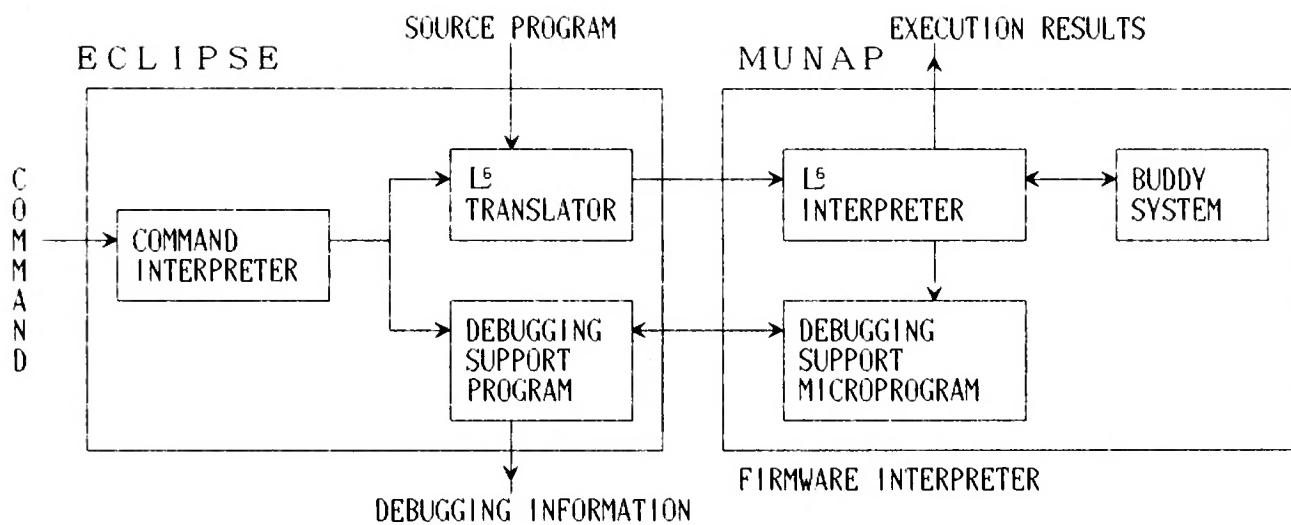


Figure 7.4
Organization of the LIDS system.

7.4.3 Debug Support Functions

Concerning the requirements for the support functions, we defined the following facilities to be implemented in the LIDS system. Notice that capital letters in parentheses correspond to user commands to use the function.

1. **Data display according to the data type (D):** Data display is a fundamental function for dynamic program debugging. As the data in a von Neumann machine are not self-identifying, the user should recognize instructions, data, and data types by him- or herself. The user should also follow a link of pointers using address data. In the LIDS system, there are five data types, namely, binary, octal, decimal, ASCII, and pointer, and a data item is always treated according to its type. For example, the contents of bugs and fields are displayed according to the types. The user can display a pointer with a pointed block and trace the

link forward and backward.

2. Range checks (T, M, S): Some languages, like PASCAL and ADA, attempt to catch as many violations of type and range constraints on data as possible at compile time. Simple examples of constraints are a limited nonnegative integer on an age, 1 to 31 on a date, and so on. However, whenever expressions are involved or data are inputted from the outside, they must be checked at run time. The LIDS system allows the following constraints on a data field: (a) data types (T), (b) maximum/minimum (M), and (c) relational expressions (S). These are to be checked automatically when data are stored in the field.

3. Detection of undefined references and dangling pointers (R, C): This function detects the failure to define the value of a data element before referencing it. It also detects the deletion of a data block while it is being referred to by other pointers. A pointer that loses the pointed block by such a failure is called a dangling pointer. These errors are checked automatically in the LIDS system. However, the user can explicitly specify a special display command for a field so that data type and value are displayed when a reading or writing operation occurs for the field.

4. Execution unit control (S, T, BRK): For the debugging of a program we sometimes want to run the program partially, inspect the result, and continue the execution. In order to meet this requirement, we provided the facilities for (a) step run (S) or trace (T) in the units of instruction, statement, and procedure,

and (b) set and reset break points (BRK).

5. **Measurement of execution frequencies (F):** It is very important to use various combinations of input data to test all the execution paths of a program. To do this, the system records the execution frequencies of program segments. Here, a segment means a program fragment that satisfies a condition that a transfer of control occurs only at the top or bottom of the fragment. The information also indicates the frequencies of true and false branches at if statements and the frequently executed parts of a program. These may be used for improving the performance of the program.

In addition to the above debug support functions, the LIDS system has such usual functions as compile (C) and run (R). **Figure 7.5** shows an example use of the commands for compiling the program of Figure 7.3a, setting break points, executing, and displaying the contents of the block that corresponds to figure 7.3b.

7.4.4 Processing in Firmware Interpreter

The interpreter fetches and executes intermediate codes according to usual instruction cycle. Frequently accessed data, such as bug, intermediate code, and descriptors for bug and field, are stored in the SPM, while the L^c program and other tables for debug support functions are stored in the MM. **Figure 7.6** shows the format of field descriptor. **Figures. 7.7 and 7.8** show the formats of the tags for bug, field, and block, respectively.

```

**THE L-SIX SYSTEM STARTS 12:34:56 04/01/86 **
COMMAND ? = C                                ; compile
SOURCE FILE NAME?=HANOI                         ; input file name
OPTION (Y/N) ? =N                           ; answer Y(yes) for recording
                                                execution frequency
COMMAND ? = BRK                            ; break command
WHAT ? = D                                 ; set break-points
      BREAK LINE NO :                         ; display present break-point (null)
      BREAK LINE NO ? =16                   ; specify break-point
      BREAK LINE NO : 16,                      ; display
WHAT ? = R                                 ; break run
DISPLAY (Y/N) ? =N                          ; no display
MOVE 1 FROM X TO Y                            ; execution result
      LINE NO : 11                           ; stop at line 11
      BREAK LINE NO : 16                      ; display break-point
WHAT ? = E                                 ; end
COMMAND ? = D                               ; display
WHAT ? = B                                 ; block
REMOTE FIELD NAME ? =A                     ; input remote field name
REMOTE FIELD NAME: A                         ; confirm the input
BLOCK ADDRESS: 00002410                       ; display block address
(ADDRESS ! FIELD NAME - TYPE !)
      0 !P-P!X-A!                         ; 0th word consists of pointer P and ASCII X
      1 !Y-A!Z-A!N-D!                      ; 1st word consists of ASCII Y and Z, and decimal N
(FIELD NAME : CONTENTS)
      P : 002408                            ; contents of defined fields
      X : X
      Y : Y
      Z : Z
      N : 2
WHAT ? = E
COMMAND ? = BYE                            ; bye command
** THE L-SIX SYSTEM ENDS 12:36:23 04/01/86 **
** GOOD BYE ! SEE YOU AGAIN ! **

```

Figure 7.5
Example of use of the LIDS system.

(bits)															
3	2	2	1	8	8	4	4	4	6	1	1	1	1	6	
DT	FS	VC	AC	AF	SB	SC	MM	BL	TP	RC	FD	TC	FO		

DT: data type
 FS: field overlap status
 VC: value check
 AC: assignment check
 AF: address of field in block
 SB: start bit position
 SC: stop condition check

MM: minimum/maximum check
 BL: byte length of field
 TP: tag position in field tags
 RC: reference check
 FD: field definition check
 TC: type check
 FO: field order in block

Figure 7.6
Field definition table.

1 (bits)		
TYPE	DEF	REF

TYPE: number, character, pointer, etc.
 DEF : defined(1), undefined(0)
 REF : referred(1), unreferred(0)

Figure 7.7
Bug and field tag.

1 6 4 8 8 (bits)				
STAT	IDEN	SIZE	#REF	#WORD

STAT: allocated(1) or free(0)
 IDEN: identification of buddy blocks
 SIZE: block size
 #REF: number of references to block
 #WORD: number of used words

Figure 7.8
Tag of block.

Using the data structures, the above mentioned support functions were implemented as follows:

1. Data display according to the data type: The data descriptors for fields and blocks are referred to in order to display the contents according to data type.

2 and 3. Range checks and detection of undefined references: The range check commands are processed by the command controller, which activates the debugging support program and the microprogram to set the specified constraints to tags, descriptors, and other tables. As shown in figure 7.9, these descriptors and tags are used to confirm the correctness of the reference. In the figure, operand ABC is accessed according to the circled numbers from 1 to 11. The symbols in the fields correspond to those of figures 7.6, 7.7, and 7.8.

4. Execution unit control: This function is implemented by replacing the next instruction address determination part of the instruction cycle with an appropriate microroutine that monitors the execution in every machine cycle and returns the control to the host when the specified condition is satisfied. The writable micro- and nanoprogram memories of MUNAP enable this replacement.

5. Measurement of execution frequencies: If this function is specified, the translator generates special intermediate codes for labeled unconditional statements and conditional statements. They have fields for recording frequencies. The recorded frequencies are collected by a debugging support microprogram,

stored in the MM, and displayed

The other functions, automatically done without user specification, include a type transformation of operand data when the types of input operands to arithmetic or logic operations differ or types of input and output operands differ.

INTER-
MEDIATE
CODE

OP	l1(3)	K1	P1 (A B C)	
----	-------	----	------------	--

BUG
DEFINITION
TABLE

①⑤③

BUG A②

TYPE	DEF	REF		CAPACITY				
------	-----	-----	--	----------	--	--	--	--

FIELD
DEFINITION
TABLE

FIELD B⑥

			AF	SB			BL	TP	FD		
--	--	--	----	----	--	--	----	----	----	--	--

FIELD C⑩

			AF	SB			BL	TP	FD		
--	--	--	----	----	--	--	----	----	----	--	--

BUG

BUG A③

BLOCK

FIELD B⑦

STAT	IDEN		#REF		
TYPE	DEF	REF			

STAT IDEN #REF

TYPE DEF REF

FIELD C ⑪

Figure 7.9
Example of field extraction through bug/field definitions.

7.4.5 Experimental Result

We made an experiment to see the effect of the support and the system performance. We assigned 9 problems to 5 students in our laboratory. The number of instructions varied from 42 to 238. Typical cases of error detection are as follows:

1. Referring to field descriptor, the system detected that the destination field was smaller than the source field of an assignment.
2. Referring to tags, the system detected a reference from an undefined bug and references to an undefined bug and field. The use of an integer as a pointer is also detected by referring to the type field of a tag.
3. Referring to a reference counter of blocks, the system detected dangling pointers.
4. Checking operand data types, the system detected an illegal arithmetic operation between a pointer and numerical data.
5. The execution frequencies were used to test execution paths and improve the performance of the program.

Consequently, 86% of the total of errors were detected by using the support functions.

The control-memory-resident microprograms for the support functions occupies 3.6% of the total of microprograms. The MM area for the functions occupies 12.3% of total area. Using five

sample programs and various commands, we made an experiment to evaluate dynamic properties. The overhead was 0.56 to 1.74% of total execution time. Thus, we can conclude that the execution time overhead is within 2% for each user command.

7.5 Firmware Implementation of Abstract Data Types

7.5.1 Data Abstraction and Relational Database Systems

The essentials of the data abstraction concept involve defining a set of data and a set of operations applicable to each data element and independent of their implementations. This encapsulation of implementation details not only attains a high degree of data independence but also enhances program productivity and maintainability. The user needs to know only the interface of the data and applicable operations. The benefits of the concept are widely recognized, and it has enjoyed many experimental applications. However, a major drawback is its performance; the abstraction requires an extra overhead to provide a data independent clear interface to the user.

Our application of MUNAP to this problem was aimed at proving the feasibility of the data abstraction technique by utilizing architectural features. The firmware/hardware combined approach was expected to enhance performance. The relational database system was selected as an application area, where data independence is one of the major objectives [OHTA84, BARO81].

7.5.2 Firmware Implementation of Physical Scheme

In relational database systems, a user query is distinguished from the physical structure of a database and the access methods (here we call these a physical schema). Therefore, information on the physical schema should be bound during the processing of the query. There are two well-known approaches. One is to translate an input query to a data independent calling sequence for generic routines. At run time the generic routine accesses the database, referring to the physical schema. We call this method the interpretative approach (I-approach). INGRES, a famous relational database system developed at University of California at Berkeley, is an example of this type [STON80]. The other is to translate an input query, referring to physical schema. Thus, the contents of the object codes reflect the schema. An example of this compilation approach (C-approach) is System R, developed at IBM [ASTR76]. While the compilation approach attains high speed performance, it lacks data independence. The interpretive approach tends to function in just the opposite way. Consequently, the usual query processing approaches that were modeled here do not essentially support both data independence and high speed performance.

In our method, called the schema conversion approach (S-approach), the physical schema is represented by pairwise definitions of "what data" exist and "what operations" are permitted, independent of a physical schema. Given queries, the translator produces an object program, referring not to a

physical schema but to a list of permitted operations for database access. The calling sequence for the selected operations is put in the object code. During execution time, the calling sequence activates the operations, defined as generic routines, which directly access the database and produce the results. This approach not only enables the object codes to be independent of the change of physical expression of data but also performs data protection for invalid database accesses, because the database is always accessed via the generic routines.

To outline the processing, we shall illustrate an example query and its processing in the S-approach. Figure 7.10 shows the data structure and logical schema for a suppliers-and-parts database [DATE82]. Figure 7.11 shows an input query, exclusive data manipulation routines, and a calling sequence of the routines. The query requests, "List tuples (S#, P#) where the parts place is the same as that of the suppliers whose status is

S	S#	SNAME	STATUS	CITY
P	P#	PNAME	COLOR	WEIGHT

RELATION	S	RELATION	P
S#	Character(6)	P#	Character(6)
SNAME	Character(20)	PNAME	Character(20)
STATUS	Integer	COLOR	Character(10)
CITY	Character(14)	WEIGHT	Integer
		CITY	Character(14)

Figure 7.10
Data structure and logical schema for suppliers-and-parts database.

larger than 6." The query is translated into a calling sequence, using 11 generic routines. The insides of the routines are omitted.

```
(* Query in SQL form *)
SELECT S#,P#
FROM   S,P
WHERE  S.CITY=P.CITY AND S.STATUS>6

(* Exclusive Data Manipulation Routines (Rs) *)
(* The routine bodies are omitted. *)
procedure INITIALIZE;
procedure POINT_FIRST_TUPLE<S>;
procedure POINT_FIRST_TUPLE<P>;
function POINT_NEXT_TUPLE<S>;
function POINT_NEXT_TUPLE<P>;
procedure MOVE<S,CITY>;
procedure MOVE_TO_RESULT<S,S#>;
procedure MOVE_TO_RESULT<P,P#>;
function COMPARE_ATTRIBUTE_><S,STATUS>:boolean;
function COMPARE_ATTRIBUTE_=<P,CITY>:boolean;
procedure CHANGE_LOG_TO_PHY<STATUS>(X:integer);

(* Calling Sequence of RS (0s) *)
begin
  INITIALIZE;
  CHANGE_LOG_TO_PHY<STATUS>(6);
  POINT_FIRST_TUPLE<S>;
  repeat
    if COMPARE_ATTRIBUTE_><S,STATUS> then
      begin
        MOVE<S,CITY>;
        POINT_FIRST_TUPLE<P>;
        repeat
          if COMPARE_ATTRIBUTE_=<P,CITY> then
            begin
              MOVE_TO_RESULT<S,S#>;
              MOVE_TO_RESULT<P,P#>;
            end;
          until POINT_FIRST_TUPLE<P>;
        end;
      until POINT_NEXT_TUPLE<S>;
  end;
```

Figure 7.11
An example of query processing by the schema conversion approach.

7.5.3 Experimental Result

In order to evaluate the degree of data independence and performance of the S-approach, we performed an experiment; the database and the query are similar to the above example, and four queries were processed on the three computers according to the three approaches. Table 7.8 shows the results. The rows are divided into four data groups for four queries, and each group is divided into three rows for the three methods. The columns are divided into three groups for three computers: MUNAP microprogram, ECLIPSE microprogram, and ACOS PASCAL. As we cannot get "microsteps" from the PASCAL program that can be compared with the data from MUNAP, we compare MUNAP with ECLIPSE by steps and MUNAP with ACOS by execution time.

1. Data independence: We assume that an attribute is changed. In the MUNAP microprograms, all 398 instructions for the C-approach, and 81 of 199 instructions for the S-approach should be changed. The changes for the S-approach are localized in 4 of 13 procedures and functions. The I-approach maintains independence during the change. This indicates the high data independence of the S-approach (compared to the C-approach).

2. Distance between the S-approach and the I- and C-approaches: The order of performance is C, S, and I for the different queries, machines, and languages. To clarify the relative distances of the three approaches, we normalize the data as follows: $X = (S-C)/(I-C)$. The results, shown in table 7.8, indicate that the S-approach is very close to the C-approach. This trend is independent of computers and languages.

Table 7.8
Performance for each query, computer, and language

Language	MUNAP microprogram				ECLIPSE microprogram		ACOS PASCAL	
Query	Execution time ^a	X %	Execution steps	X %	Execution steps	X %	Execution time ^a	X %
1	C	0.7		1,047		1,396		6.5
	S	1.0	6.7	1,578	7.0	2,551	8.3	24
	I	5.4		8,679		15,234		260
2	C	0.7		1,062		1,214		5.4
	S	1.0	6.9	1,630	7.2	2,545	10.8	20
	I	5.6		8,907		13,603		170
3	C	146.2		232,265		301,328		1,520
	S	216.0	8.4	348,849	8.8	555,688	14.1	5,840
	I	980.3		1,557,381		2,108,649		17,100
4	C	25.4		40,377		52,286		260
	S	37.6	8.3	60,605	8.7	96,760	14.0	930
	I	171.4		272,267		369,632		3,500

a. In milliseconds .

3. Overhead in the S-approach: The drawback of the S-approach is the increase in overhead for calling procedures. This overhead changes depending on the machine and language. It is the greatest in the ACOS PASCAL, next in the ECLIPSE microprograms, and the least in the MUNAP microprograms for all queries. The hardware stack operations in MUNAP greatly contributed to this tendency.

Consequently, we can say the S-approach is made feasible by the firmware/hardware support of MUNAP.

7.6 Prolog on MUNAP

7.6.1 Prolog and Parallel Processing

Prolog is a powerful and elegantly simple logic programming language with implementations ranging from mainframe to microprocessors [TICK84]. The first interpreter was developed at the University of Marseille in 1973. The compiler, which was developed at the University of Edinburgh, performed 15 to 20 times faster than the Marseille interpreter. The machine code, called PLM, has been utilized as a basis for various implementations. In spite of numerous such implementations, the goals for future inference engines require computing faster than the speed of light. This implies the need to do things concurrently.

To illustrate possible parallelism in Prolog, we shall use a simple example program, shown in figure 7.12. The definitions here mostly follow [WARR77]. The program consists of clauses. Each clause comprises a *head* and a *body*, followed by a period. The head and body are separated by delimiter ":-." The body consists of a sequence of zero or more goals (or *Procedure calls*). The head and goals of a clause are all examples of *terms*, each of which comprises a *functor* and a list of one or more terms called *arguments*. For example, the compound term

Clause No.	Clause	Meanings
1	male(mike).	Mike is male.
2	male(jeffrey).	Jeffrey is male.
3	female(carol).	Carol is female.
4	female(lisa).	Lisa is female.
5	child(lisa,mike).	Lisa is a child of Mike.
6	father(X,Y) :- male(X), child(Y,X).	X is the father of Y if X is male, and Y is a child of X.
7	?-father(Who,lisa).	Who is the father of Lisa?

Figure 7.12
An example Prolog program.

`child(lisa,mike)` has functor "child" of arity two, with arguments "lisa" and "mike." The identifiers beginning with small and capital letters indicate constants and variables, respectively. A clause that comprises head, body, or both head and body is called a *fact*, *question*, or *rule*, respectively. In the example, clauses 1 through 5, clause 6, and clause 7 correspond to fact, rule, and question, respectively. A sequence of clauses whose heads all have the same functor is called a *procedure*. In the example, clauses 1 and 2 form a procedure with functor 'male'. The depth of nesting of a term in a clause is specified by a *level number*. The head and goals of a clause are at level 0; their immediate arguments are at level 1; and so on for levels 2, 3, etc. A compound term not at level 0 is called a *skeleton term*.

To find an answer to the question `father(Who, lisa)`, the system searches for the first clause whose head *matches* or *unifies* with it. The match is found at clause 6, and Who and lisa are bound to X and Y, respectively. The matching clause

instance is then activated by executing in turn, from left to right, each of the goals of its body. In the example, goals 'male' and 'child' are executed in this order. Thus, the unification of `male(X)` and the first clause, `male(mike)`, bind 'mike' to X. The next unification of `child(lisa,mike)` with the fifth clause succeeds and 'mike' is outputted as an answer. If at any time the system fails to find a match for a goal, it *backtracks*. That is, it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause, and tries to find a subsequent clause that also matches the goal. Execution terminates successfully when there are no more goals waiting to be executed. In the example, the recursive search occurs if the clauses 1 and 2 are exchanged (readers are encouraged to try several cases).

Note that the above example indicates several possible parallelisms. We shall list three potential parallelisms.

AND Parallelism: When a clause has multiple goals, they may be activated in parallel. In clause 6 of the example, two goals, `male(X)` and `child(Y, X)`, may be activated in parallel. Note that the same value should be bound to two Xs. This consistency condition causes a lot of overhead for this parallelism.

OR parallelism: When a procedure includes multiple clauses, they may be executed in parallel. In the example, procedures for 'male' and 'female' have this parallelism. The AND and OR parallelisms may be combined to obtain higher performance.

Unification parallelism: When a goal has multiple arguments, they may be unified in parallel. For example, goal "child" in the example has two arguments, Y and X, and the matching with the corresponding atoms "lisa" and "mike" may be done in parallel. Note that the effect of this parallelism depends on the number of arguments, and a consistency check (e.g., on AND parallelism) is necessary.

Considering the parallelism, we decided to concentrate on the implementation of the unification parallelism. The reasons are as follows:

1. The register-transfer level parallel architecture of MUNAP seems to be appropriate for investigating the effect of the unification parallelism.
2. If we can prove its effectiveness, the parallel unification unit can be designed based on the results, and may be used not only for a sequential Prolog processor but also for the AND/OR parallel processor as a component unit that performs the unification function.
3. Compared to AND and OR parallelism, the unification parallelism has not attracted much attention, and the effect has not been clarified experimentally.

7.6.2 Parallel Processing of Unification

The efforts to increase parallel processable streams may be

made at compile time and execution time. We shall describe in outline our ideas [INAG87]:

1. Decomposition of skeleton terms: At compile time, when a head includes a skeleton term, we decompose it into a functor and the arguments. This decomposition increases the number of arguments to be processed in parallel. For example, the following head is assumed to be processed in four processor units:

```
concatenate(cons(X,L1), L2, cons(X,L3)):- {goals .....}
```

It includes two skeleton terms "cons" at level 1. **Figure 7.13** shows that the two functors are unified in parallel at the first stage, and the four arguments (X, L1, L3 at level 2 and L2 at level 1) are unified in parallel at the second stage. To enable uniform access to the arguments at different levels, we introduced a stack, called an argument stack. It keeps the pointer pairs from which the argument address may be accessed.

2. Clustering: As mentioned earlier, the parallel unification may bind different values to the same variable. A method for avoiding the overhead for the consistency check is to assign the same variables to the same processor when they appear in a head. **Figure 7.14** illustrates the effect of clustering for an example unification of the following head,

```
head(V1, V2, V3, V2):- .....
```

where V1, V2, and V3 are variables. The second and fourth arguments are the same. If we unify the variables in four processors separately, as shown in figure 7.14a, the consistency check is necessary after the unification. If the two V2s are clustered in PU#1, they are processed sequentially and the consistency check becomes unnecessary. Thus, by losing the parallelism for the same variables, the clustering reduces the overhead for the consistency check. We decided to cluster the variables at compile time to avoid the run time overhead.

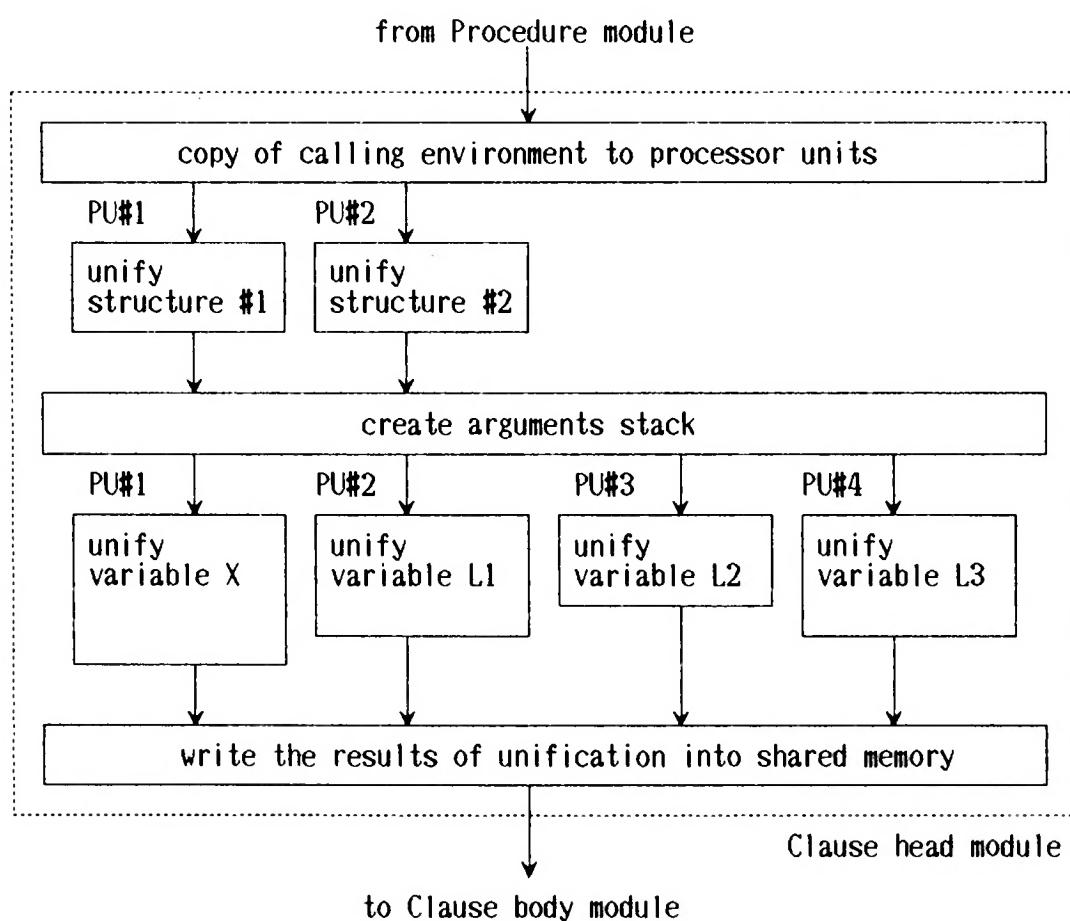


Figure 7.13
Parallel unification of structures and arguments.

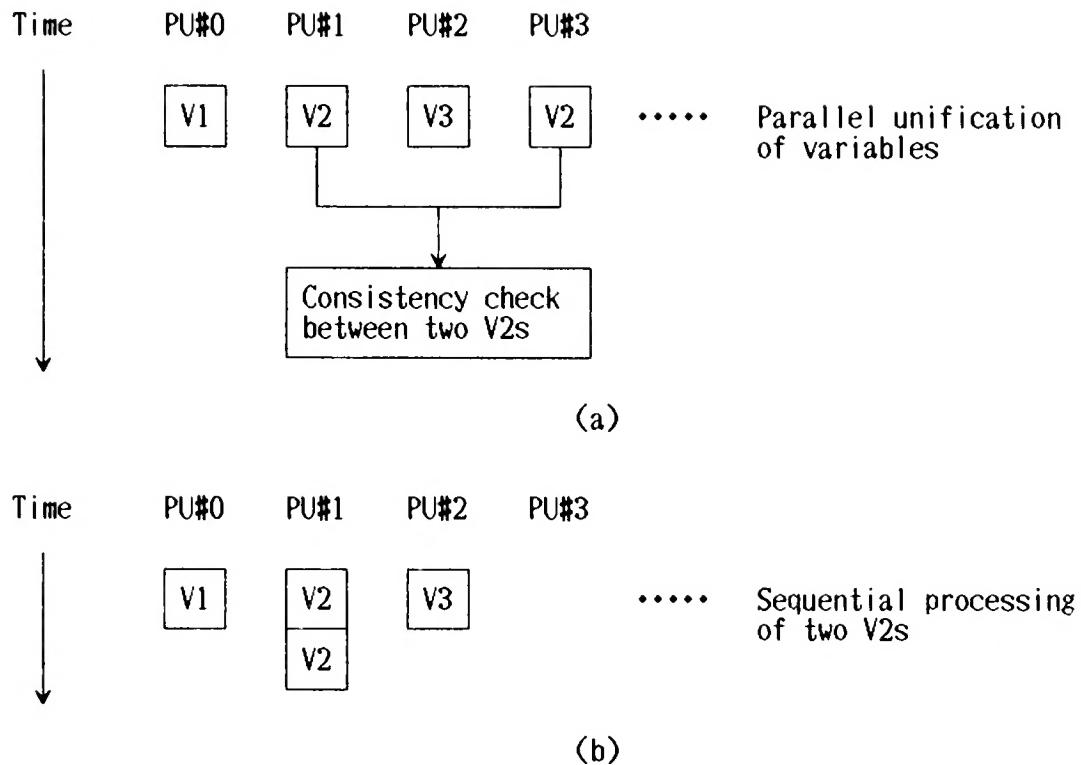


Figure 7.14
Parallel unification and consistency preservation:
(a) consistency check; (b) clustering.

3. Run time consistency check: The compile time clustering leaves a possibility of inconsistency, caused by subgoals. For example, the following unification of subgoal with a head requires consistency between the first and second arguments:

```
?-likes(X, X, Y). .... subgoal (question)
likes(A, B, C). .... head
```

In this problem, we place the same subgoal variables (in this case X) in the common memory. When a process wants to update the value, it should check on whether the value is undefined or the assigned value is the same as the value of the process.

4. Extension of PLM: To facilitate the parallel processing of arguments, we defined our own PLM codes, called **PLMI**. The level of PLMI is higher than that of PLM codes in that multiple PLM codes were synthesized to one PLMI code to enhance the parallelism within an instruction processing. Further, two instructions were newly defined to implement the unification parallelism. They are the "unify-arguments" and "unify-structures" instructions, which unify multiple variables and literals in a head and multiple structures in a head, respectively.

7.6.3 Firmware Interpreter on MUNAP

We have developed a PLMI interpreter on MUNAP to evaluate the effectiveness of our ideas. The compiler decomposes skeleton terms and performs clustering. The interpreter fetches, decodes, and executes PLMI. We shall now describe the data structure and the outline of processing of the interpreter.

1. Data Structures: The most important decision concerning the data structures is the decomposition of all the data into local memory in the PU (i.e., SPM and RF) and global common memory (i.e., MM). As the key is to allow parallel processing of unification, the necessary data for the unification are allocated to the local memory.

The local memory includes (i) registers used as an instruction counter and several stack pointers, (ii) values of the subgoals' arguments, (iii) the argument stack generated by structure unification, and (iv) functor literals.

The global memory includes (i) PLMI codes except for functor literals, (ii) a local stack for local variables, (iii) a global stack for global variables, (iv) a molecule stack for structure sharing [WARR77], and (v) a trail stack for backtracking.

2. Processing: The interpreter consists of four major modules, named "procedure," "clause head," "clause body," and "backtrack," as shown in figure 7.15. Given a question (subgoal), the interpreter first initializes the system. The clause body module reads in argument values for the subgoal from the variable stack and broadcasts them to multiple PUs. The procedure module

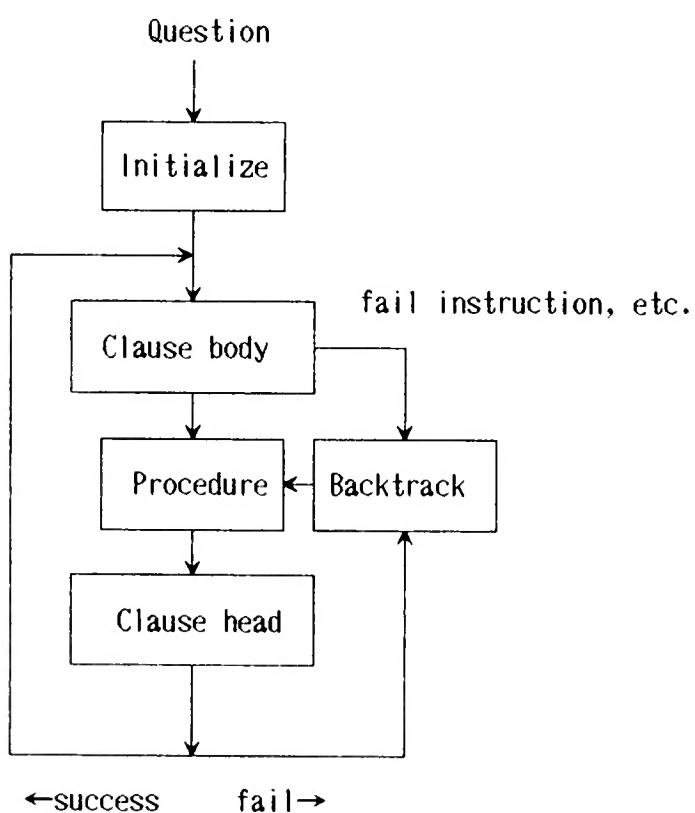


Figure 7.15
Control flow of firmware interpreter.

selects a clause from the corresponding procedure for the subgoal. Then the control goes to the procedure head module, which performs the parallel unification of the arguments included in the head of the selected clause. The last module realizes the parallel algorithm shown in figure 7.13. The other modules essentially perform the task serially, while utilizing the low level parallelism of MUNAP.

3. Utilization of the Micro-Nano Interaction Mechanism: Compared to the other projects, the Prolog project utilizes the micro-nano interaction mechanism (described in section 3.4.1) very effectively for a low level distributed processing. These operations are to be done under the partial join mode.

Interaction for parallel unification: The nanoprocesses for parallel unification are activated by micro and run in parallel. The micro watches the unification nanoprocesses, which is indicated by the messages put on port registers. Thus, the micro goes the rounds of four port registers while nanoprocesses are doing unification. As the success of the unification requires the success of all the arguments being unified in the multiple nanoprocesses, the failure of an argument makes the other continuing nanoprocesses meaningless. In this case, the micro is informed of failing nanoprocess, it stops the other running nanoprocesses, and the control goes to the backtrack module. Below, the effect of this low level process switching will be described on the basis of an experiment.

Interaction for global memory access: During the

unification, the multinanoprocesses need to access global data in the MM. In this case, a nanoprocess sends a request to the micro. The micro accesses the MM and passes the results to the nano.

7.6.4 Experimental Result

To evaluate the effectiveness of our idea and MUNAP's architectural features, we described five programs in Prolog and processed by the firmware interpreter. The programs were (i) N-reverse for reversing 25 list elements (NREV), (ii) Quick sort for sorting 50 data items (QSORT), (iii) Tower of Hanoi for 4 disks (HANOI), (iv) Permutation for generating permutations of 4 integers (PERM), and (v) Eight queens until getting the first solution (QUEEN).

The static features with respect to arguments are summarized in table 7.9. The average number of arguments after clustering (i.e., 3.2) indicates the degree of possible parallelism.

From detailed dynamic data we obtained the following observations:

1. **Effect of unification parallelism:** To clarify the effect, we compared the execution steps of the unification part, obtained from unification parallel processing on MUNAP, with that for a single PU configuration without unification parallelism. The results in table 7.10 indicate that on average 41% of steps were saved by the parallel operations.

Table 7.9
Number of arguments^a

Program	Procedure	A	B	C	D
NREV	nreverse	2 2	1 0	3 2	3 2
	concatenate	3 3	2 0	5 3	4 2
QSORT	qsort	3 3	1 0	4 3	4 2
	partition	4 4 4	2 2 0	6 6 3	5 5 3
HANOI	start	2	0	2	2
	hanoi	4 4	1 1	6 6	6 6
	append	3 3	1 2	3 5	2 4
PERM	perm	2 2	0 1	2 3	2 3
	delete	3 3	1 2	4 5	2 4
QUEEN	queen	1	0	1	1
	put	3 3	0 0	3 3	2 3
	select	3 3	1 2	4 5	2 4
	not-take	2	0	2	2
	not-take1	3 3	0 1	3 4	3 4
Average		2. 9	0. 8	3. 7	3. 2

a. key: A, original number of arguments; B, number of skeleton terms; C, number of arguments after the decomposition of skeleton terms; D, number of arguments after clustering.

Table 7.10
Effect of unification parallelism at the procedure head module

	NREV	QSORT	HANOI	PERM	QUEEN
4PU/1PU	0.58	0.61	0.69	0.61	0.48

The effect may change depending on the number of processors.

Figure 7.16 shows that the improvement is marked from 1 to 2 PUs, and gradual from 2 to 4 PUs. This result indicates that the unification parallelism is suitable for a processor with small parallelism.

2. Effect of clustering: The compile time clustering made consistency checks unnecessary. The reason may be that the need for a consistency check arises from the variables included in a head because, when we unify a subgoal and the corresponding head,

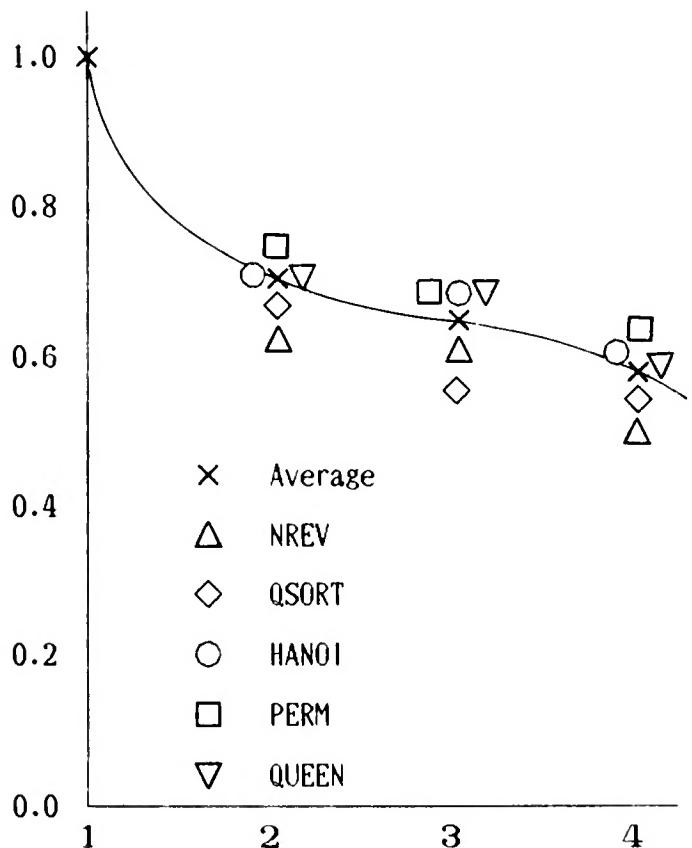


Figure 7.16
Number of processor units versus execution steps of procedure body module: vertical axis, ratio of execution steps to that of the single processor; horizontal axis, number of processor units.

the variables in subgoals are usually bound to constants, while the variables in the head are undefined.

3. Effect of the micro-nano interaction mechanism: As mentioned earlier, a nanoprocess failing in unification stops the other running nanoprocesses. The effect varies from 0.5 to 6.2% depending on the nature of the programs. The effect is large when a long nanoprocess runs until the end without being notified of the failure of the other nanoprocess.

4. Ratios of execution steps: As shown in table 7.11, the Procedure head and Procedure body modules consume most of the processing time. And even after applying parallel unification, the unification process in the Procedure head module still occupies 53 to 75% of the total of execution steps. This proved the feasibility of our effort to perform the unification in parallel as well as the necessity of further work to improve the performance.

Table 7.11
Ratios of execution steps at five processing modules

Program	Procedure head	Procedure body	Procedure	Backtrack	Initialize
NREV	73.4	22.9	3.3	0.4	0.0
QSORT	68.9	25.1	2.9	3.1	0.0
HANOI	75.4	16.6	3.8	4.2	0.0
PERM	67.8	18.6	5.0	8.6	0.0
QUEEN	53.3	39.6	3.6	4.1	0.0
Average	67.6	24.6	3.7	4.1	0.0

The effect of the architectural features of MUNAP will be described in chapter 8.

7.7 Smalltalk-80 on MUNAP

7.7.1 Smalltalk-80 and Parallel Processing

The Smalltalk-80 system's roots go back more than ten years, to in the Xerox Palo Alto Research Center. The system is distinguished by the visual impact of bitmapped graphics on highly interactive user interfaces and by increased flexibility in terms of user programmability [KRAS83]. The virtual machine is completely specified by [GOLD83], in which the required low level behavior of any Smalltalk-80 implementation is described. However, it is pointed out that the direct translation of the Smalltalk-80 methods of model implementation may exhibit disappointing performance levels. The specification may not be optimal for a particular host computer.

Many efforts have been made to attain acceptable performance within a specific environment [KRAS83]. Recent architectural approaches include SOAR, Sword32 and Tektronix 4404. SOAR [UNGE84a] was developed on the Smalltalk oriented chip of RISC type. It employs tagged 32-bit object pointers (Oop) to reduce the overhead for data type check and uses a unique garbage collection mechanism, called Generation Scavenging [UNGE84b]. Sword32 [SUZU84] is also a special purpose chip with 256 general purpose registers. Tektronix 4404 is a commercially available machine that implements several techniques in order to enhance

performance. However, this single processor architecture does not utilize language processing parallelism sufficiently.

Our major objective in applying MUNAP to Smalltalk-80 is to exploit implicit parallelism by using multiple processors. As the level of the byte code virtual interface was too low to utilize implicit parallelism, we defined a higher level intermediate code, called I-code (Intermediate code) [DOI87]. We have also been trying to utilize architectural features of MUNAP when implementing the interpreter.

Before looking into our firmware interpreter, we shall briefly introduce the language and its standard implementation techniques [GOLD83].

7.7.2 Smalltalk-80 and Its Implementation

1. **Language:** The basic concepts of an object oriented language are an *object* and a *message*. The object represents components, such as numbers, character strings, dictionaries, etc. Each object consists of some private memory and a set of operations. The message is a request for an object to carry out one of its operations. A message expression, such as "A mes: B," describes a receiver "A," selector "mes," and arguments "B," For example, the message expression

1 + 2

asks the receiver "1" to calculate and return the sum. The additional object 2 is an argument that specifies the amount to be added. If a message expression includes an assignment prefix,

such as

```
sum <- 1 + 2
```

the object returned by the receiver will become the new object referred to by the variable.

The implementation of a set of objects that all represent the same kind of system component is defined as a *class*. The individual objects described by a class are called its *instances*. All instances of a class have the same message interface. An object's private properties are a set of *instance variables* that make up its private memory and a set of *methods* that describe how to carry out its operations. A class includes a method for each type of operation its instances can perform. The operations of some methods, called *primitive methods*, are predefined and built into the virtual machine. The classes form a tree structure; *subclass* (*superclass*) stands for a class that inherits (*bequeaths*) variables and methods from (to) an existing class.

To show some of the language features, figure 7.17 illustrates a sample program that tests whether the characters in a given string are symmetric [DOI86]. Figure 7.17a shows that the class name is *Symmetry* and the method "judge" is defined in it. After getting the center position of the given string by the computation of (aList size//2), it compares the characters from both ends. The repetitive comparison is realized by the "1 to num do;" structure, which corresponds to the "for i=1 to num do" loop of PASCAL. Figure 7.17b illustrates a method of executing "judge" after setting string "ABCBA" to string instance "list."

```

class name Symmetry
superclass Object

instance methods
  testing
    judge:aList
    {
      | num i j |
      num ←aList size//2.
      i ←0.
      j ←aList size.
      1 to num do: [:k | (aList at:i)=(aList at:j)
                           ifTrue: [i←i+1. j←j-1]
                           ifFalse:[ ↑false]].
      ↑ true
    }
  
```

(a)

```

x ←Symmetry new.
list←String new:5.
list at:1 put:'A'.
list at:2 put:'B'.
list at:3 put:'C'.
list at:4 put:'B'.
list at:5 put:'A'.
result←x judge:list
  
```

(b)

Figure 7.17
 Example Smalltalk-80 program:
 (a) class definition of "Symmetry"; (b) execution
 for string "ABCBA".

2. Implementation: Language implementation realizes a virtual machine specified in the book by Goldberg et al. [GOLD83]. It consists of two major parts: *Interpreter*, which fetches and executes 256 byte code instructions, and the *object memory manager*, which provides the interpreter with an interface to the objects that make up the Smalltalk-80 virtual image. The

implementation of the interpreter includes the realization of primitive methods. The object memory and interpreter communicate about objects with object pointers (Oop).

3. The compiler and interpreter: Source methods written by programmers are translated by a compiler into *compiled methods*, and byte codes in the compiled methods are interpretively executed by the interpreter. The essential role of the interpreter is the execution of methods. Figure 7.18 shows a message passing state caused by a byte code in the compiled methods [DOI86]. The method context represents the state of the execution. It points to the instance of the receiver; the receiver has a pointer to a class; the class points to the method dictionary. If the method cannot be found in the dictionary, search continues to the dictionary of the superclass. Thus, the overhead of the message passing process is one of the major problems.

4. Memory management: Garbage collection is a popular technique for managing free space. However, the interactive programming environment of the Smalltalk-80 requires a special technique that will enable users not to have to wait while collecting garbage. Delayed garbage collection and generation scavenging are two well-known algorithms for this purpose [UNGE84b].

7.7.3 Definition of a New Code for Parallel Processing

As mentioned earlier, our objective was to exploit parallelism for operations with large overhead, such as message passing. To

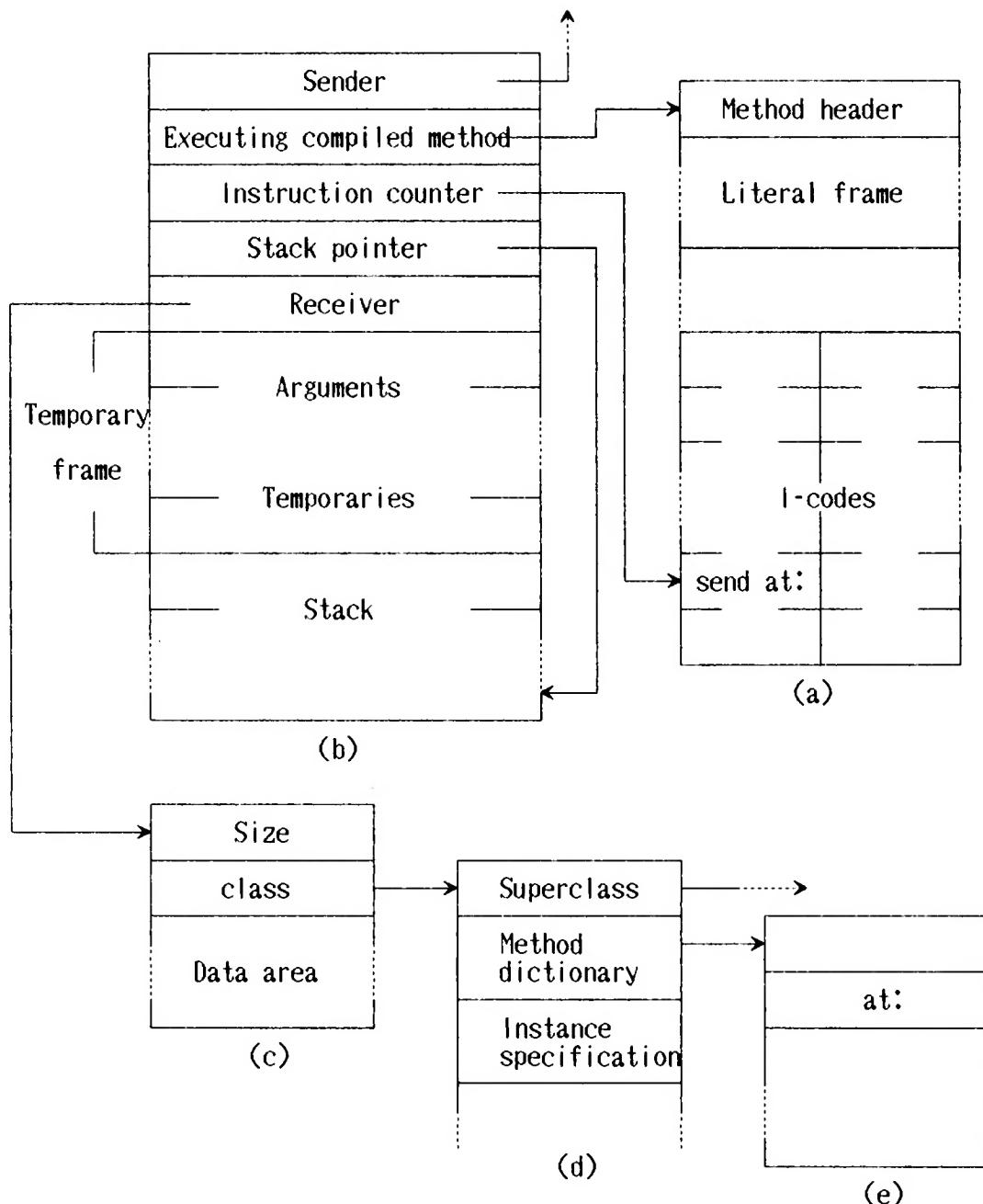


Figure 7.18
Execution status of a method: (a) compiled method; (b) method contexts;
(c) instance (receiver); (d) class; (e) method dictionary.

realize this, however, the functional decomposition at byte code level is not appropriate. Thus, we defined a higher level, new intermediate codes, called I-code [DOI87]. The format of the I-code is shown in figure 7.19. The lengths of Send and Special-send instructions are variable, while those of Jump and Return instructions are fixed at 2 bytes. The multiple object fields (OF) in the former two instructions allow parallel processing of the arguments. We will briefly describe the instructions.

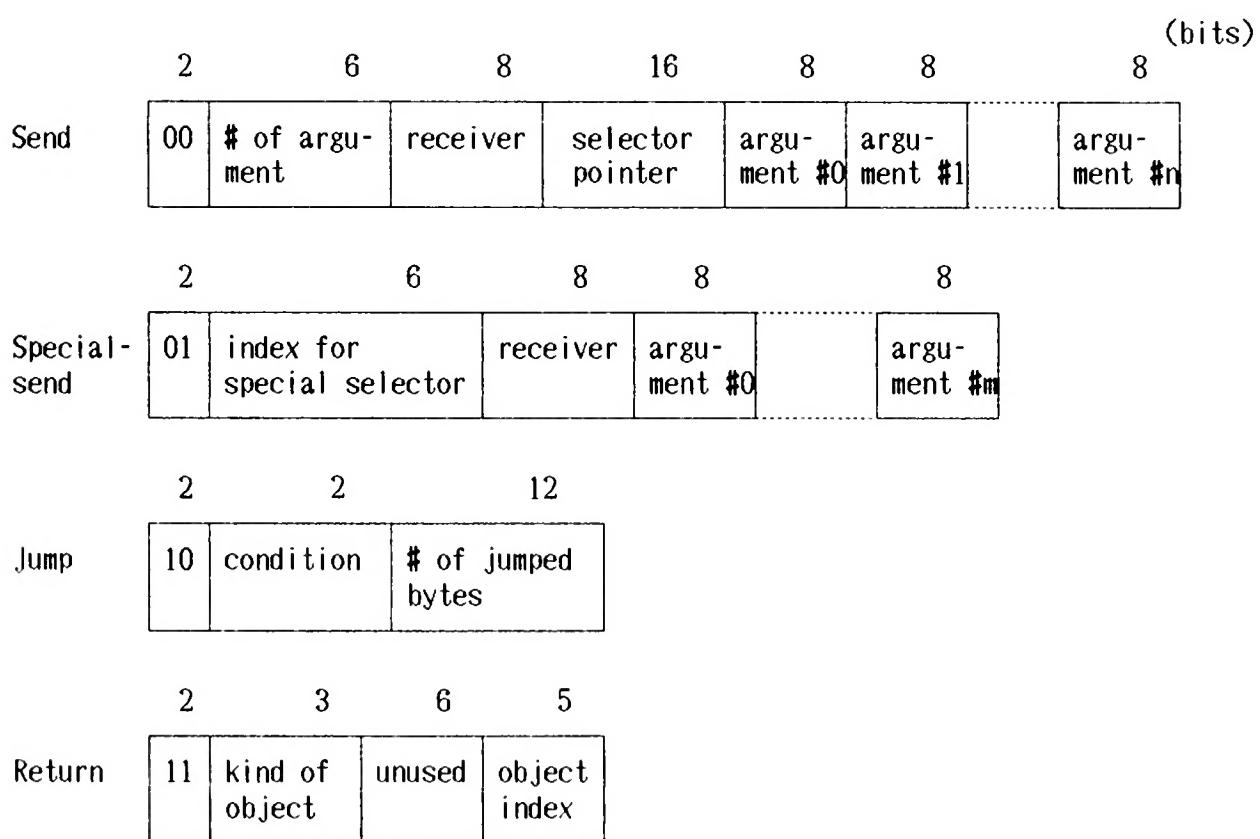


Figure 7.19
I-code instructions.

Send instructions consist of the number of arguments: OF for receiver, selector pointer for message selector, and OFs for arguments. While the byte code indirectly accesses a selector pointer, the I-code can directly access it.

Special-send instructions cover frequently used messages with 0 to 2 arguments and save memory space. They include arithmetic/logic operations, access to array and character strings, and instance generation.

Jump instructions make conditional and unconditional branches.

Return instructions return a control from the executing method to the calling one with a return value. Temporary frame and instance variables may be specified for a return value (This cannot be specified in byte codes).

Basically, the functions of I-codes are richer than those of byte codes. The correspondence between the I-code and the byte code is summed up in **table 7.12**.

Table 7.12
Correspondence between I-code and byte code

I-code	Byte code
Send	Send (0 - 3 arguments) + Push Oop
Special-send	Send (arithmetic and special) + Push Oop
Assignment	Push & Pop Oop, Set variable
Jump	Jump
Return	Push Oop & Return

7.7.4 Firmware Interpreter for the I-Code

The interpreter fetches an I-code instruction and dispatches to 22 routines according to the most significant 5 bits of the instruction. The division and concatenation unit (DCU) extracts the 5 bits and places them on the OPR port register, a port from nano to micro. Adding the 5 bits on the port register with a base address value, the B microinstruction makes a functional branch to a branch table with 32 entries, where 32 unconditional branch B microinstructions are stored. This process realizes dispatching in 3 steps. In the routines, the semantics of the instructions are implemented. The following are the unique features of the processing:

1. Parallel Oop determination: Unlike the byte code interpreter, the I-code interpreter determines two Ooops in parallel. The reason for the small parallelism is based on the observation that the number of arguments in a message sending instruction is usually not so large. R. Meyers reported that 90% of the instructions have less than 2 arguments [KRAS83]. Thus, in our routine, all the combinations of two Ooops for 5 frequently used OFs are processed in parallel. Twenty-five parallel Oop determination microroutines have been developed for this purpose.

2. Parallel search in method dictionary: The dictionary contains selectors to be accessed through hash function as shown in figure 7.20 [DOI86]. The message "mesB" is broadcast to four PUs. A hash function is defined as the logical AND of the selector pointer and the size of the hash space. The parallel comparison

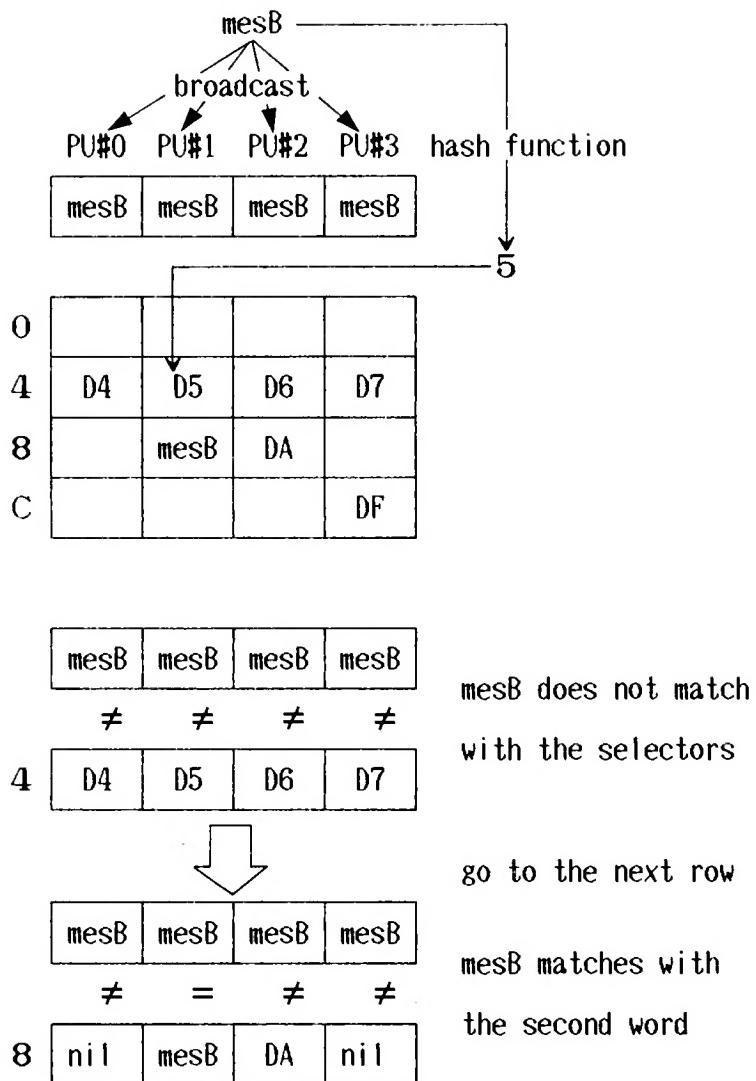


Figure 7.20
Parallel search of method dictionary.

starts from the row that includes the address for the hashed value. If "mesB" is not in the row, the succeeding rows are scanned in the same manner.

3. Tagged object pointer: An object pointer (Oop) has 4-bit tag field to enhance the performance of class determination and garbage collection. The tag indicates a kind of literal (Smallinteger, Character, Boolean, or UndefinedObject), a pointer

to objects created at compile time (pointer to class or pointer to literal instance), a pointer to contexts (pointer to method context or pointer to block context), or pointers to objects created at run time (pointers to objects in old space or pointers to objects in present or future space). The last item is used for the following garbage collection.

4. Garbage collection: We have adopted the Generation Scavenging algorithm. It divides a free memory space into three areas, called old space, present space and future space. Long life objects are placed in the old space. The present and old spaces include the other objects and interchange their roles by copying the effective contents of present space to future space. The tag bits are used to distinguish Oops for this purpose.

7.7.5 Experimental Results

In order to evaluate the effectiveness of the I-code and the firmware interpreter, we have developed the byte code interpreter on MUNAP and compared it with the I-code interpreter on MUNAP [DOI87]. Five programs were developed on both the I-code and byte code interpreters as follows:

- a. *Cache memory simulation:* 7 stores and 2 fetches are done for a cache memory.
- b. *Eight-queens problem:* It finds n solutions for the eight queens. Here, we stop the program after finding the first solution.

- c. *Binary tree operation*: It makes 5 insertions and 2 deletions for a binary tree.
- d. *Drunken cockroach*: A cockroach moves on 3 x 3 tiles according to generated random number.
- e. *Symmetry*: It tests whether a given character string is symmetric or not, as shown in figure 7.17.

Programs (a) from Fuchi and Suzuki [FUCH85], and programs (c) and (d) are from Goldberg et al. [GOLD83]. Programs (b) and (e) were newly described for the experiment [DOI86].

1. **Evaluation of I-Code:** Instead of long instruction I-code lengths, the memory space for the I-code compiled methods are 1.01 of that for the byte code ones. This indicates that instructions with appropriate functions will not increase the memory space. The comparison of the instruction execution steps is shown in table 7.13. The number of executed I-code instructions is 0.47 of that of byte code ones. The major reason is that the stack instructions in byte codes are unnecessary for I-code because the function is replaced by the specification of OF fields of I-code instructions. This reduces the overhead of instruction cycles.

2. **Firmware Interpreter:** The I-code and byte code interpreters need 1.5 K micro- and 3.4 K nanoinstructions, and 1.6 K micro- and 2.8 K nanoinstructions, respectively. In the I-code

Table 7.13
Comparison of dynamically executed instructions

		a	b	c	d	e ^a
I-code	Total steps	4,016	14,803	203	2,910	58
	Ratios of executed instructions					
	Send	1.7	0.8	23.7	12.6	1.7
	Special-send	54.6	53.6	25.1	42.0	51.7
	Assignment	11.5	7.1	12.8	13.4	22.4
	Jump	21.1	30.7	11.3	14.9	15.5
	Return	11.1	7.8	27.1	17.1	8.6
Byte code	Total steps	8,924	31,635	363	5,296	120
	Ratios of executed instructions					
	Send	25.6	25.8	27.5	31.0	27.5
	Pop & store	4.5	6.3	8.8	8.9	11.7
	Jump	9.8	14.7	6.6	8.9	8.3
	Return	5.0	36.7	15.2	9.4	4.2
	Push	50.1	49.5	41.6	42.5	48.3

a. a - e represent the sample programs described in section 7.7.5.

interpreter, the Oop parallel determination routine occupies a large space, while, in the byte code interpreter, the dispatch table for 256 routines occupies a large space. The total microsteps for I-code programs are 0.81 of those for byte code programs, as shown in table 7.14. This is due to a smaller number of instruction execution cycles and the enhanced parallelism. This is evidenced by the fact that the number of

Table 7.14
Dynamically executed microsteps

	a	b	c	d	e
I-code (A)	145,750	505,311	11,447	135,344	2,863
Byte code (B)	182,014	626,193	14,986	163,871	3,076
A/B	0.80	0.81	0.76	0.83	0.93

average active PUs is 2.57 for I-code and 2.38 for byte code. Table 7.15 shows the ratios of execution microsteps in the I-code interpreter modules. Except for Primitive methods, the Oop determination and Fetch & dispatch routines consume nearly half of the steps. The ratio of 7% to 10% for Send and Special-send instruction routines is higher than for the other instruction dependent routines, such as Assignment, Jump, and Return.

3. Effect of Parallel Oop Determination Routine: 43-100 % (average 69%) of instructions with multiple arguments utilize this parallelism. The average execution microstep improvement by this routine was 0.88. The adequacy of MUNAP control memory space suggests that the routine should cover all the combinations between eight kinds of objects.

4. Effect of Parallel Method Dictionary Search: This resulted in a 4% improvement in execution steps. It is small. However, the step-by-step inspection of the routine suggested that as the number of the keys in the table increases, the effect of the parallel scanning will become clearer.

Table 7.15
Ratios of microsteps for I-code interpreter modules

Function	a	b	c	d	e
Fetch & dispatch	18.5	20.0	15.5	16.8	16.5
Oop determination	27.1	27.2	10.8	15.7	26.0
Send, Special-send	9.1	10.1	6.9	7.7	8.7
Assignment	5.5	3.6	5.7	6.1	1.4
Jump	2.3	4.1	1.3	1.7	3.8
Return	4.5	3.6	14.9	13.4	9.2
Receiver's class determination	2.5	2.6	5.4	5.0	3.6
Method dictionary search	0.8	0.1	12.2	2.5	11.0
Method cache search	2.2	2.3	4.7	3.7	3.9
Context creation & update	0.9	1.0	12.4	7.8	1.9
Method header processing	1.1	1.1	6.3	6.2	1.6
Primitive methods	25.3	24.1	3.9	13.4	12.4

5. Effect of Multiprocessor Architecture: To evaluate the effect, we computed the execution steps of byte code interpreter on a single PU. The ratio of the steps for the I-code on MUNAP, those for the byte code on MUNAP, and those for the byte code on a single PU MUNAP is 0.47:0.56:1. The differences between 0.47 and 0.56 and between 0.56 and 1 indicate the effect of I-code definition and the effect of multiprocessors, respectively. This indicates that the MUNAP architecture greatly contributes to performance improvement not only for I-code but also for byte code.

7.8 Three-Dimensional Color Graphics

7.8.1 Graphics and Parallel Processing

Computer graphics is of rapidly growing importance in the computer field as an extremely effective medium for communication between man and computer. In particular, the display of three-dimensional objects and scenes is involved in many computer graphics applications. For example, computer aided design systems allow their users to manipulate models of machined components and automobile bodies, and simulation systems present a continuously moving picture of a three-dimensional world to the pilot of an aircraft [NEWM79]. Producing a realistic image on a two-dimensional screen, however, presents many problems. These involve, among other things, the basic techniques for modeling and transformations as well as the techniques for achieving realism in the display of depth, hidden surface elimination, shading, and so on. A growing number of algorithms have been developed to address these problems [FOLE82].

It is natural that these technological advances require extensive processing capabilities. The size of the number of pixels to be processed is also a source of difficulty. For example, a 512 x 512 screen includes 262,144 pixels. If we apply an operation to the pixels uniformly, the cost becomes proportional to the number of the pixels. The most promising way to cope with this problem is to utilize parallel and pipeline processing. For this purpose, special purpose processors have been proposed and implemented.

7.8.2 Application to MUNAP

To evaluate the effectiveness of the MUNAP architecture in this field, we decided to develop a three-dimensional color graphics system on MUNAP [SUZU85]. We found that the L^c language may be used as a basis. We were encouraged by the fact that several graphics processing algorithms seem to be based on a linked list data structure [NEWM79]; the topological relationship between faces, edges, and vertices to be displayed may be represented by a linked-list data structure. Figure 7.21 illustrates components of faces, edges, and vertices connected via pointers. In addition to the topological information, geometric information should be represented in a way that allows a variety of transformations, such as rotation, scaling, and viewing. They are usually implemented through 4 x 4 matrix operations. Figure 7.22 shows three matrices for translation, rotation, and scaling. As the matrices are independent of the points to be transformed,

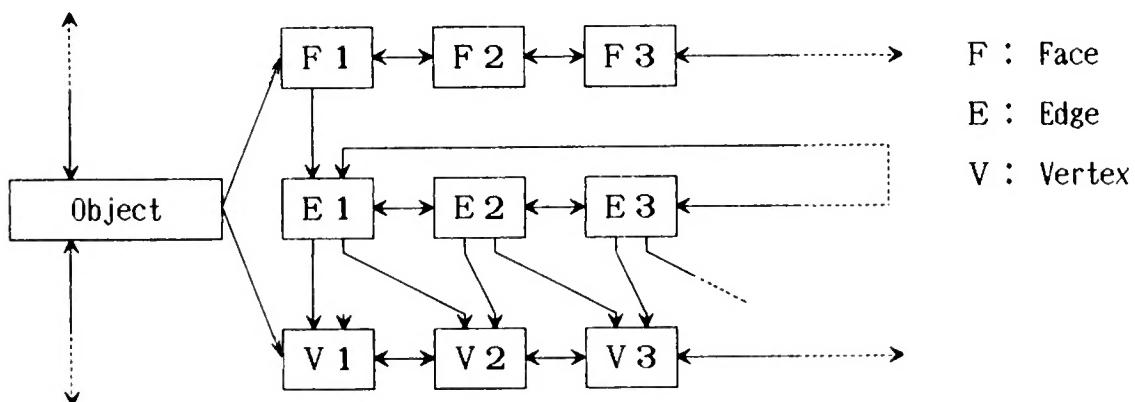


Figure 7.21
List structure for object data.

$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$ <p>Ti : components of translation</p>	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ <p>θ : angle</p>	$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ <p>Si : ratio</p>
(a)	(b)	(c)

Figure 7.22
Transformation matrices: (a) translation; (b) rotation (about the x coordinate axis); (c) scaling.

the concatenation of those transforms may be represented by the product of the corresponding matrices. Note that the components of the transformation matrices include elementary functions, such as trigonometric functions. In applications where the viewpoint changes rapidly or where objects move in relation to each other, transformations must be carried out repeatedly. It is therefore necessary to find efficient ways of performing three-dimensional transformations. To achieve realism, we are faced with many candidate algorithms [NEWM79, FOLE82]. Our present choice is a polygon clipping algorithm by Weiler and Atherton for hidden surface elimination and shadow generation. Figure 7.23 illustrates the clipping process. Along the numbered arrows, the two areas, (1,2,3,4) and (9,10,11), of the object polygon are clipped by the shaded clipping polygon. This operation may be implemented by the list traversal and insertion operations of the L⁶ language. Based on these considerations, we extended the L⁶ so that it includes

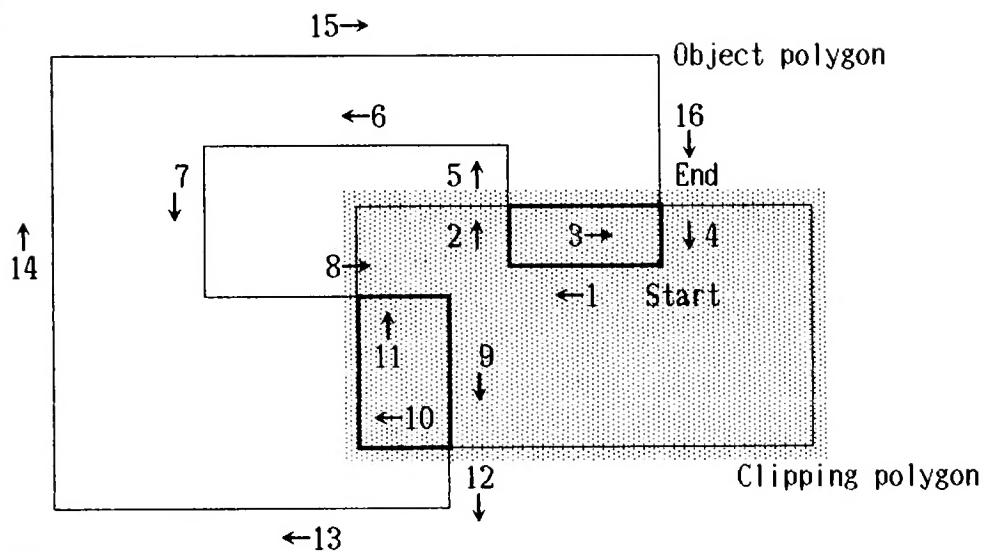


Figure 7.23
Polygon clipping.

1. matrix operation for 1×4 and 4×1 vectors and for 4×4 matrices with 16-bit floating point components,
2. elementary functions, such as trigonometric and square root functions, and
3. reading and writing operations for pixel data.

Adding up these functions and the original L^c functions, we redefined the L^c language and developed its interpreter through the MUNAP interpreter. A monitor display was attached to MUNAP so that the pixel data generated and written into the extended MM of MUNAP can be displayed on it. The refresh buffer consists of a 512×512 8-bit pixel area. The 8 bits are divided into 3, 3, and 2 bits for red, green, and blue, respectively. We know this display part is not very powerful. However, it is enough for our objective of clarifying the effect of the MUNAP architecture.

7.8.3 Firmware Interpreter for the Extended L^c

The basic part for the original L^c had been developed when we started this project. Thus, our main job was to develop microprograms for the extended part and embed them into the original ones.

1. Matrix Operation: It is convenient that the four PUs correspond to the dimension of homogeneous coordinates for three-dimensional graphics. For example, the product of a 1×4 vector (a, b, c, d) and a 4×4 matrix A requires 16 multiplications and 12 additions (i.e., 4 times of 4 multiplications and 3 additions). The four PU parallel operation of MUNAP reduces them to one-third. An inner product of two vectors (a, b, c, d) and (e, f, g, h) may be evaluated as shown in figure 7.24. At first, each of four PUs has (a, e) , (b, f) , (c, g) , and (d, h) , respectively. The first multiplication makes 4 products. Then, they are concurrently exchanged between PUs 0 and 1, 2 and 3, in parallel. (For the SEN exchange operation, see figure 3.2c.) The results are further added, exchanged, and added as shown in the figure. This process reduces 4 multiplications and 3 additions to 1 multiplication and 2 additions. Note that the results are obtained in four PUs in parallel.

2. Elementary Function Evaluation: We used CORDIC [WALT71] for trigonometric function evaluation. The method is suitable for microprogram implementation because it evaluates by the

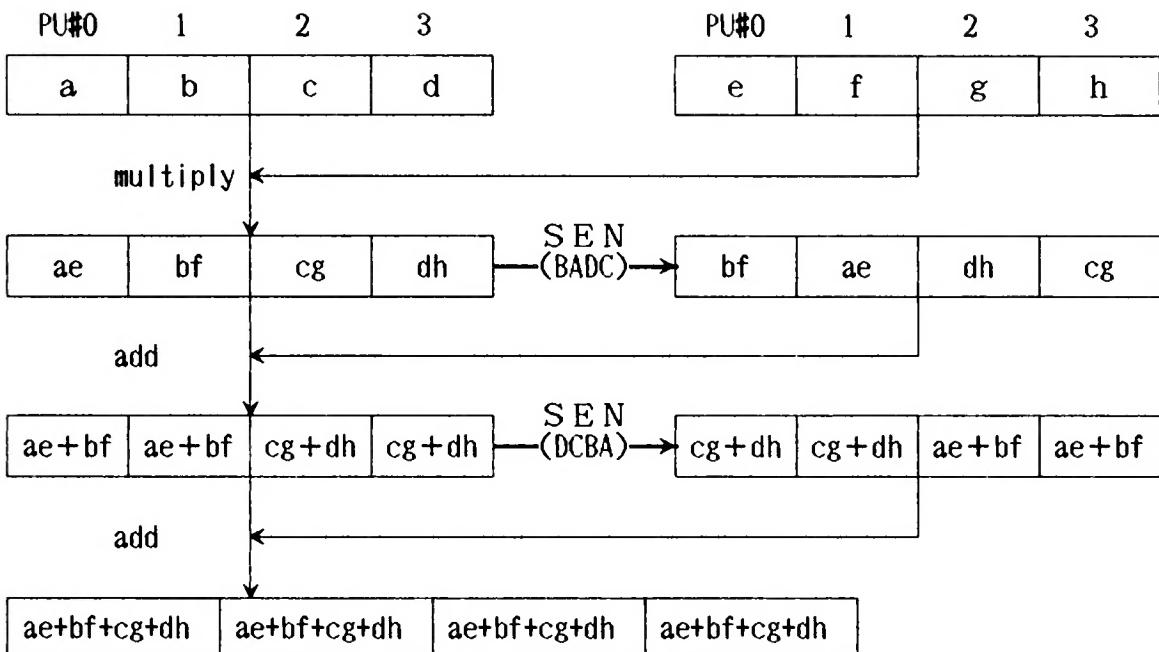


Figure 7.24
Inner product of two vectors.

repetitive use of addition, subtraction and shift. The necessary constants are stored in the SPM. The square root function is evaluated by a restoring method. Throughout the evaluations, the bit operation function is utilized for appending bits and shifting data.

3. Pixel Data Processing: RGB(Red, Green, and Blue) elements of pixel data are handled in parallel in three PUs, and the results are concatenated to produce 8-bit data. DCU and BOU functions are utilized to support the bit operations. Eight 8-bit data are wrapped up and written into refresh buffer memory, which is developed as an extended MM of MUNAP.

7.8.4 Experimental Results

Statically, the extended 36 instructions were implemented by 1.2 K microinstructions and 4 0.8 K nanoinstructions. This resulted in a 24% increase in control memory space.

As to dynamic evaluation, we cannot make a total evaluation, because the graphics system is under development, using the extended L⁶. However, we obtained preliminary results from test programs written in the extended L⁶ for simple two-dimensional graphics operations. These include such operations as drawing a point, a line, and a circle. The results show that (i) each operation was executed by about 500 to 750 L⁶ instructions, (ii) each instruction required about 300 microsteps, and (iii) the average number of active PUs was 2.4. For the matrix operations, 3.2-3.5 PUs were active. This indicates that the parallelism of MUNAP is appropriate for processing a pixel data.

We are now trying to develop the total graphics system on MUNAP; at its completion, we shall be able to evaluate the total effect of the language extension and the supporting architecture.

7.9 Numerical Computation for Large Amounts of Data

7.9.1 Why Numerical Computation on a Nonnumeric Oriented Machine?

The primary goal of the MUNAP is to explore nonnumeric oriented application areas that have relatively less explicit parallelism than numerical ones, such as array processing. Thus, it might sound strange to apply such a machine to numerical computation when a large amount of explicit parallelism is required. The

reasons for the application are as follows:

1. In our machine, the term "nonnumeric oriented" is based on the enrichment of nonnumeric functions in addition to ordinary functions. Thus, it might be better to call the machine a universal host computer that has various types of functions and a wide range of applications. We should learn the limitations of this universality by applying the machine in a seemingly unsuitable area.
2. The present MUNAP has parallelism by four processor units, although the architecture is expandable to control multiples of two processor units. The parallelism of four PUs will be enough for some areas, but it may not be for others. It is important to know the limitations of the parallelism by applying it to an area with explicit parallelism.

We selected two typical problems [SHIB85]: Fast Fourier Transform (FFT) [BRIG74] and the LU decomposition for simultaneous linear equations [ARNO83].

7.9.2 Fast Fourier Transform

The discrete Fourier transform is defined by the following equation:

$$X(n) = \sum_{k=0}^{N-1} x_0(k)e^{-j2\pi nk/N}, \quad (7.1)$$

where $n=0, 1, \dots, N-1$. Let $W = e^{-j2\pi/N}$. Then the equation may be rewritten using matrix notation as follows:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^9 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}. \quad (7.2)$$

The equation may be decomposed into a product of two simpler matrices:

$$\begin{bmatrix} X(0) \\ X(2) \\ X(1) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & W^0 & 0 & 0 \\ 1 & W^2 & 0 & 0 \\ 0 & 0 & 1 & W^1 \\ 0 & 0 & 1 & W^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^0 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^2 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}. \quad (7.3)$$

Notice that the order of elements $X(i)$ on the left-hand side is changed. It is denoted $\overline{X(n)}$. Thus, the computation is divided into these two stages:

$$\begin{bmatrix} x_1(0) \\ x_1(1) \\ x_1(2) \\ x_1(3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^0 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^2 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}, \quad (7.4)$$

$$\begin{bmatrix} X(0) \\ X(2) \\ X(1) \\ X(3) \end{bmatrix} = \begin{bmatrix} X_2(0) \\ X_2(1) \\ X_2(2) \\ X_2(3) \end{bmatrix} = \begin{bmatrix} 1 & W^0 & 0 & 0 \\ 1 & W^2 & 0 & 0 \\ 0 & 0 & 1 & W^1 \\ 0 & 0 & 1 & W^3 \end{bmatrix} \begin{bmatrix} X_1(0) \\ X_1(1) \\ X_1(2) \\ X_1(3) \end{bmatrix}. \quad (7.5)$$

These two stages may be executed sequentially, as shown in figure 7.25. At each node, using two data elements, inputted from the previous stage, coupled with a preassigned value of W , the output is computed and broadcast. This basic operation is called *butterfly*. Now we have reached the key of the FFT algorithm. Our task is to implement the data flow in MUNAP for a large number of data elements.

We made an FFT microprogram for $N = 1024$. A data element is in the 48-bit wide floating point representation. As the MM consists of eight banks of 8-bit memory, we allocated the 48-bit

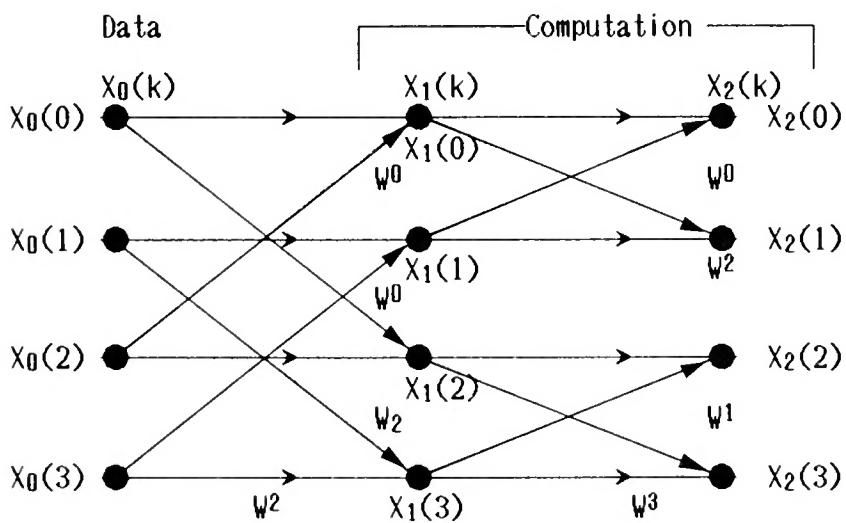


Figure 7.25
Data flow diagram for FFT, $N=4$.

data elements in consecutive 64-bit words in the MM. The method does not leave an unallocated area, and it requires an extra operation to write data partially (48-bit within 64-bit) from four PUs. Starting from the stored data in the MM, the microprogram performs FFT by loading them, doing butterfly operations, and storing the result. Figure 7.26 illustrates how four data elements in the MM are read into SPMs and distributed to four PUs so that a data element is folded into three consecutive addresses in the SPM.

- a. Access four elements from MM to SPM0-3 so that their first 16 bits come to PU0,1,2, and 3, respectively. Transfer the first elements to the first 16-bit input area of SPM(30) through the shuffle exchange network.
- b and c. Shift and transfer the second and third elements to SPM(31) and SPM(32), respectively by using the network.

This transfer method requires one-fourth as many steps as the sequential one. The storing process is the reverse of this process.

The input data are processed by butterfly operations. The operation includes two major operations to be performed in parallel, namely, sine and cosine functions and bit reversal.

The sine/cosine function may be implemented in three ways: (1) polynomial equation evaluation, (2) CORDIC [WALT71], or (3) table lookup. The first method is time consuming because it requires multiplications. The second method includes simple

	PU0	PU1	PU2	PU3
SPM20	0,0	0,1	0,2	
SPM21		1,0	1,1	1,2
SPM22	2,2		2,0	2,1
SPM23	3,1	3,2		3,0

	PU0	PU1	PU2	PU3
SPM30	0,0	1,0	2,0	3,0
SPM31				
SPM32				

(a)

	PU0	PU1	PU2	PU3
SPM20	0,0	0,1	0,2	
SPM21		1,0	1,1	1,2
SPM22	2,2		2,0	2,1
SPM23	3,1	3,2		3,0

	PU0	PU1	PU2	PU3
SPM30	0,0	1,0	2,0	3,0
SPM31	0,1	1,1	2,1	3,1
SPM32				

(b)

	PU0	PU1	PU2	PU3
SPM20	0,0	0,1	0,2	
SPM21		1,0	1,1	1,2
SPM22	2,2		2,0	2,1
SPM23	3,1	3,2		3,0

	PU0	PU1	PU2	PU3
SPM30	0,0	1,0	2,0	3,0
SPM31	0,1	1,1	2,1	3,1
SPM32	0,2	1,2	2,2	3,2

(c)

Figure 7.26
Distribution of matrix elements to four PUs: (a) first transfer;
(b) second transfer; (c) third transfer.

arithmetic operations, such as addition, subtraction, and shift, and uses a table for storing $\arctan 2^{-i}$, where i is an integer. Method (2) is 2 to 5 times faster than method (1). Clearly, method (3) is the fastest of the three, but requires the largest memory area for the table. To cover angles from 0 to $\pi/2$ by steps of $2\pi/1024 = \pi/512$, we need the following words, provided that each item occupies 1.5 word:

$$1.5 \times (\pi/2) / (\pi/512) = 384 \text{ (word)}. \quad (7.6)$$

As the words can be stored in the SPM for fast access, we chose this method. Further, we provide a buffer area for W because the same W s are repeatedly used in the same or consecutive stages.

The bit reversal function may be implemented in either of the following three ways in MUNAP: (1) table lookup, (2) a combination of the SEN mirror transformation and table lookup, and (3) repetition of bit test and set. Method (1) directly accesses a reversed bit string by table lookup. Method (2) transforms an input by the SEN 4-bit unit mirror transformation and table lookup for 4 bits. Method (3) tests an input word bit by bit and sets the bits of an output word according to the test result. In order to compare the methods, we made the microprograms and evaluated them. Table 7.16 shows the result: (1) is high speed but requires a lot of SPM area; (3) is slow. Thus, we used (2) for our FFT microprogram.

Table 7.16
Three implementations of bit reversal function

	Table reference	Mirror transform and table reference	Bit test and set
Machine cycles	3	12	76
Micro- and nanowords (bits)	136	1,096	536
SPM area (words)	1,024	4	0

7.9.3 Experimental Result for FFT

Table 7.17 shows the result of experimentation. The total steps are divided into three phases, and the butterfly phase is further divided into several parts, as shown in **table 7.18**. The butterfly phase occupies 90%. This indicates the effectiveness of parallel operations in this phase. In the butterfly, multiplication is the largest overhead. Thus, hardware multipliers are desirable if we apply MUNAP here. The buffer for

Table 7.17
Analysis of Fast Fourier Transform microprogram

Function	Steps	Ratio (%)
Control	64,511	6.0
Data exchange	51,422	4.8
Butterfly	1,066,289	89.2
Total	1,182,222	100.0

Table 7.18
Details of the butterfly

Operation	Ratio (%)
Load/Save	20.4
Add/Subtract	22.6
Multiply	45.1
Sine/Cosine	3.7
Others	8.2

the trigonometric function has a 90% hit ratio and contributes to a reduction by 28,000 steps.

We further considered the effect of the number of PUs. Figure 7.27 shows that the number of steps decreases in proportion to the number of PUs. However, it saturates because the control and exchange phase cannot be enhanced by the parallelism.

7.9.4 LU Decomposition for Large Sparse Matrix

The LU decomposition is a method for solving simultaneous linear equations. An n -dimensional equation is formalized as follows:

$$A\mathbf{x} = \mathbf{b}, \quad (7.7)$$

where A is $n \times n$ matrix, \mathbf{x} , and \mathbf{b} are n -dimensional vector. The

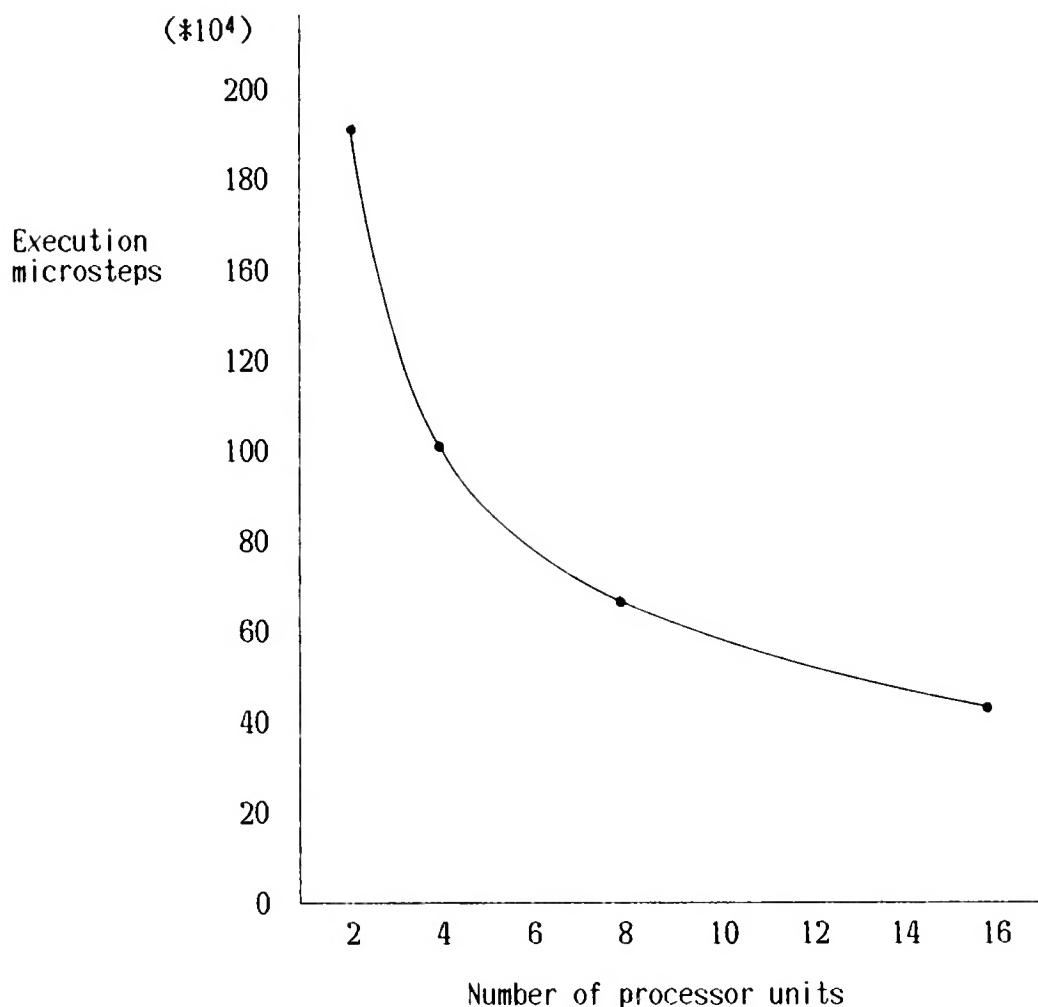


Figure 7.27
Number of processor units versus execution microsteps.

matrix may be decomposed into a product of two matrices.

$$A = LU, \quad (7.8)$$

where L has nonzero elements on the diagonal line and below the line; and U has 1's on the diagonal line and nonzero elements above it. Let

$$U\mathbf{x} = \mathbf{y}. \quad (7.9)$$

Then

$$L\mathbf{y} = \mathbf{b}. \quad (7.10)$$

Equation (7.10) can be solved by the forward substitution method, i.e., computing $y[1], y[2], \dots, y[n]$ in this order. Equation (7.9) can be solved by backward substitution method, i.e., computing $x[n], x[n-1], \dots, x[1]$ in this order.

Thus, the LU decomposition consists of the following four phases:

1. *Normalization*: Normalize the elements row by row, by dividing them by the diagonal element.
2. *Forward deletion*: Make matrices L and U by repeating multiplication and subtraction.
3. *Forward substitution*: Compute \mathbf{y} .
4. *Backward substitution*: Compute \mathbf{x} .

We applied the method to a large sparse matrix. The nonzero elements, generated for a sparse matrix by the above matrix computation, are called *fill-in*. The minimization problem of such elements is beyond the scope of this book [ARN083]. The length of an element is determined to be 24 bits, i.e., 7 for the exponent and 16 for the mantissa. We defined two sets of microprograms for the real and complex numbers. The operations

include addition, subtraction, multiplication, and division. In order to store a large sparse matrix, we considered the following conditions:

- a. The element for a given row and column numbers should be accessed quickly.
- b. The L and U elements in a row should be accessed at a time.
- c. Fill-ins should be treated in the same way as the other elements.

Several configurations were considered, and the data structure, shown in figure 7.28 was selected. Figure 7.29 shows the corresponding equation.

The key to the application is to find parallel processible operations. The natural way to treat this kind of problem is by a loop distribution method. We applied it to the above mentioned four phases. There are two possiblities: row wise or column wise parallelism. We named the row wise one *RLU*, the combination of row wise and column wise ones *CLU*. In the latter case, the sparse matrix is decomposed into several sections, and row wise or column wise parallelism is selected in each section considering the positions of the nonzero elements.

7.9.5 Experimental Result for LU Decomposition

We made two microprograms, i.e., *RLU* and *CLU*. We defined three matrices. The following is one of the three examples we defined

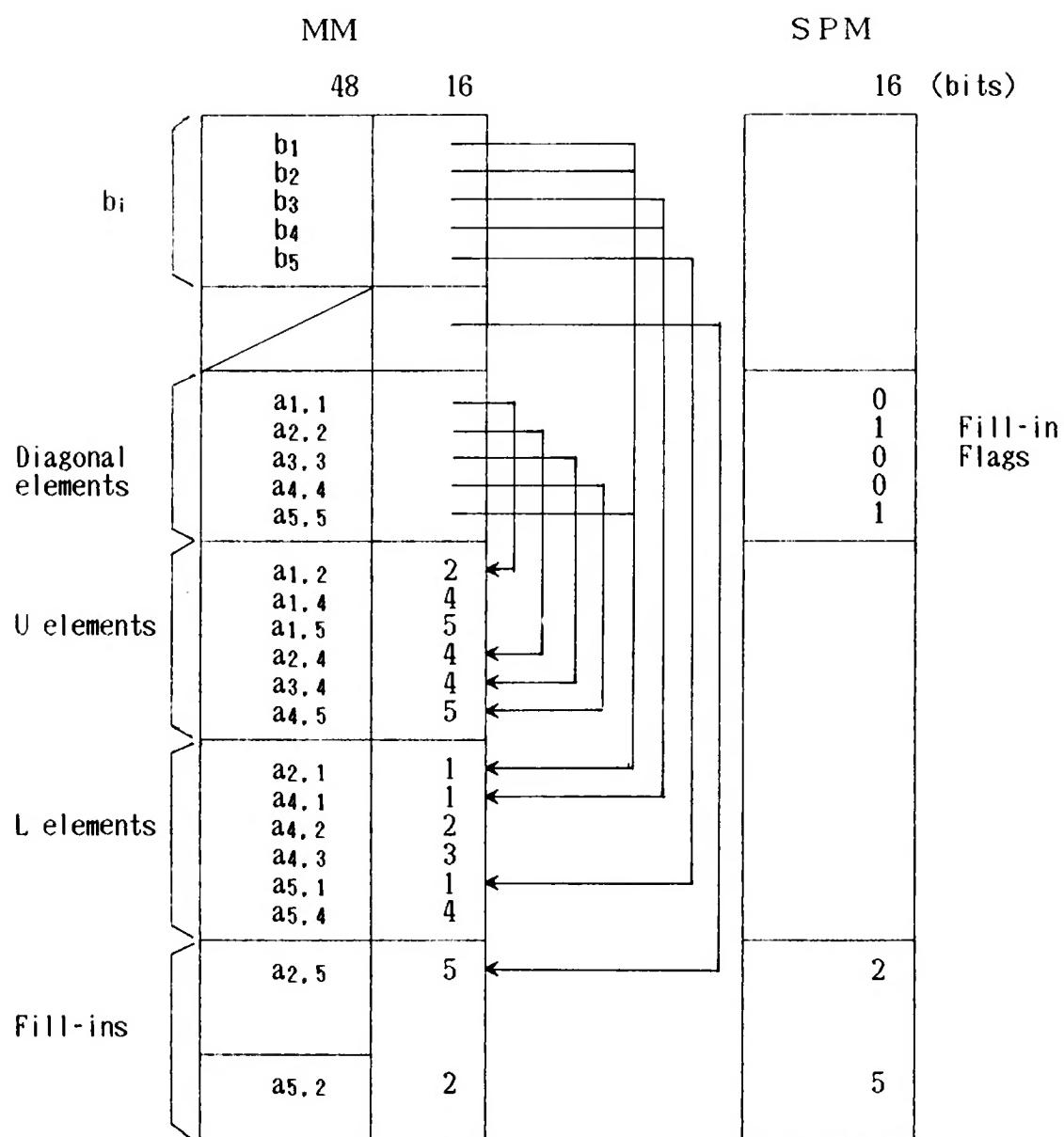


Figure 7.28
Memory allocation for the example equation.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & 0 & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & 0 & a_{2,4} & 0 \\ 0 & 0 & a_{3,3} & a_{3,4} & 0 \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ a_{5,1} & 0 & 0 & a_{5,4} & a_{5,5} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}$$

(a)

$$\begin{bmatrix} a_{1,1} & a_{1,2} & 0 & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & 0 & a_{2,4} & a_{2,5} \\ 0 & 0 & a_{3,3} & a_{3,4} & 0 \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ a_{5,1} & a_{5,2} & 0 & a_{5,4} & a_{5,5} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix}$$

(b)

Figure 7.29
An example equation: (a) initial state; (b) after fill-in.

and used for the experiment. The matrix is 150 x 150 with 320 nondiagonal, nonzero elements. The number of fill-ins is 48. The sparse ratio is 97.7%.

Table 7.19 shows the execution steps of an RLU microprogram for the four phases. The steps for each phase are divided into control and computation parts. The total number of microsteps is 631,578, which means 348.7 milliseconds for the execution time. Using the evaluator we obtained 3.6 for the average PU utilization. This is a fairly high usage ratio. However, further investigation of the computation part shows that 1.28 PUs were doing real arithmetic operations, such as addition, subtraction, multiplication, and division. This is because of the small number of nonzero elements in each column.

Table 7.20 shows the result for a CLU microprogram. This employs row wise parallel operations until the 109th row and columnwise parallel operation after the row. The total number of

Table 7.19
Rowwise LU decomposition microprogram

	Control	Computation	Total (ratio)
Initialize	2,692	0	2,692 (0.4%)
LU decomposition	51,754	214,559	226,313 (42.2%)
Forward substitution	18,551	221,012	239,563 (37.9%)
Backward substitution	22,159	100,851	123,010 (19.5%)
Total	95,146	536,432	631,578

Table 7.20
Columnwise LU decomposition microprogram

	Control	Computation	Total (ratio)
Initialize	2,693	0	2,693 (0.8%)
LU decomposition	30,935	109,271	140,206 (41.9%)
Forward substitution	14,992	116,647	131,639 (39.4%)
Backward substitution	13,939	45,784	59,723 (17.9%)
Total	62,559	271,702	334,261

steps becomes 334,261, which means 184.8 milliseconds for the execution time. The average PU number, which does the real computation, is enhanced to 2.5. Thus, the CLU method is shown to be good for a sparse matrix with a small number of nonzero elements in each column.

Next, we shall look into the effect of the increase in PUs. Figure 7.30 shows this effect by hand simulation. The curves show that RLU is good for a less sparse matrix and CLU is good for a sparse matrix. The sequential nature of the last two phases (i.e., forward and backward substitution) prevents the application of parallel machine vector processing. In the first and second phases (i.e., normalization and forward deletion),

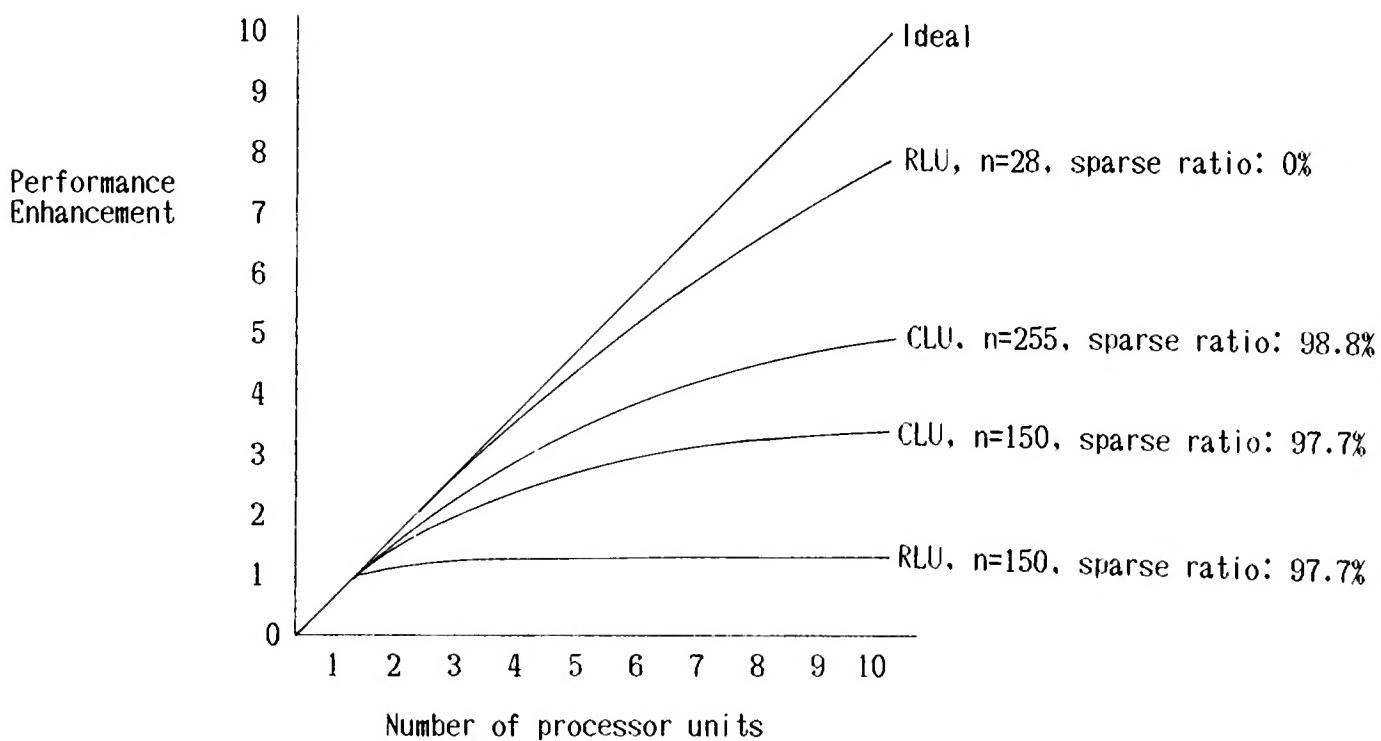


Figure 7.30
Number of processor units versus performance enhancement for LU decomposition.

irregular distribution of nonzero elements makes the effective vector operations difficult. This leads to the conclusion that the vector operations for the sparse matrix cannot enhance performance in a way that is proportional to the number of PUs; the register level parallelism of MUNAP is appropriate for a matrix with high sparse ratio.

Architectural Evaluation and Improvement

8.1 Evaluation as a Universal Host Computer

A universal host computer originally was defined as a machine that can be microprogrammed to emulate any desired target machine. However, this notion is perhaps restrictive, especially if the target machine is perceived as a machine language programmable computer. A great deal of work has already been done in microprogrammable machines for the direct execution of higher level languages, microprogramming of operating systems, and even direct microprogramming of applications programs [SHIB80, SHIB82, FURU82]. In this sense, our applications of MUNAP to several areas can be regarded as an experiment to evaluate the amenability of a universal host to various environments.

In this section, we shall first select typical results from

the applications and evaluate MUNAP architecture based on the investigation in Yamazaki et al. [YAMA86a]. In other words, the evaluation here is MUNAP oriented, unlike the application oriented evaluation in chapter 7. The data we selected are from the following areas:

- a. Emulator of a minicomputer ECLIPSE (section 7.2).
- b. MUNAP System Description Language Interpreter (section 7.3).
- c. Low Level List Processor (section 7.4).
- d. Prolog Interpreter (section 7.6).
- e. Smalltalk-80 Interpreter (section 7.7).
- f. Fast Fourier Transform (section 7.9).

Table 8.1 summarizes the test programs and their execution microsteps. The results of investigation are described in sections 8.2 to 8.4 according to the design principles.

The next concern is directed to the improvement of the architecture. In section 8.5, we shall try to find a systematic way to achieve architectural improvement and apply it to the experimental results. The method is based on the idea of architectural synthesis and may be applicable to any other machine.

8.2 Parallelism of Multiple Processor Units

8.2.1 SIMD versus MIMD Parallelism

When multiple processor units are activated, they are called of SIMD or MIMD type if all the activated nanoinstructions

Table 8.1
List of problems

Area	Programs	Execution steps
Emulator	Integral computation	73,218
	Church's problem	45,354
	Graph search	132,366
	Topological sort	46,838
	Tower of Hanoi	71,771
L ^b	Tree manipulation	157,553
	String comparison	120,920
	Graph search	587,791
	Topological sort	162,112
	Tower of Hanoi(3)	158,095
MSDL	Integral computation	707,305
	String permutation	33,119
	Church's problem	205,295
	Bit string manipulation	122,614
	Tower of Hanoi(4)	272,262
Prolog	Nreverse(25)	115,061
	Eight queens	968,035
	Tower of Hanoi(3)	16,179
Smalltalk	Cache	145,750
	Eight queens	505,311
	Binary tree	11,447
	Drunken cockroach	135,344
	Symmetry	2,863
Numerical computation	FFT	1,182,222
	LU decomposition	631,578

are, respectively, the same or not. The ratios between SIMD and MIMD are 1:4, 3:7, 1:1, 1:1, 1:5, and 9:1 for the ECLIPSE emulator, MSDL, L^c, Prolog, Smalltalk, and FFT, respectively. Globally speaking, emulations of a target machine and languages require high percentages of MIMD type processing, while the numerical computation mostly utilizes SIMD type processing.

8.2.2 Average PU Utilization

Our basic question was, How many processors were running for each microcycle? The answer is 2.2, 2.2, 2.4, 1.8, 2.5, and 3.8 for the ECLIPSE emulator, MSDL, L^c, Prolog, Smalltalk, and FFT, respectively. Looking into the first five applications, we found that the uniformity of variables assignment is very important for the enhancement of the parallelism. That is, a group of variables that will not be likely to be accessed at the same time should be allocated to the same PU; and the difference in the total numbers of variables in multiple PUs should be small. A basic word length also affected them. Some of the example programs used 64-bit words (for example, floating point numbers of the ECLIPSE emulator). That raised the average number a little. In FFT, four independent nanoprogram streams attain high utilization of the parallelism.

8.2.3 Effect of Serial Operations

The serial operation, defined in subsection 3.4.2, was intended to provide operations of multiprocessor units, not only in parallel but in series, i.e., the serial execution of mutually

dependent operations of four PU's on the source data. Table 8.2 shows the effect. The number of stages indicates the number of PUs involved in a serial operation. The number of operations indicates the number of PUs in which a numerical or nonnumeric operation was performed against the input data. Thus, if data items A and B are added in PU0 and the result is transferred and stored into PU1, the serial operation is 2 stages and 1 operation. The results show that (a) operations of 2-stage, 1-operation were best used, (b) operations of 2- and 3-stage, 2-operation, were used next, (c) 4-stage operations were rare, and (d) the operations were not used in FFT because the PUs are mutually independent and do not need to exchange data. The use of the 4-stage operation in Smalltalk-80 is for calculating the

Table 8.2
Average ratios of serial operations

Number of stages	Number of operations	Numerical computation					
		Emulator	L ^s	MSDL	Prolog	Smalltalk	
2	1	20.1	3.4	19.5	0	12.1	0
	2	1.8	0.7	3.3	0	19.0	0
3	1	0	0	1.3	0	0	0
	2	0	0.7	5.1	0	0.4	0
	3	0	1.3	0.3	0	2.6	0
4	2	0	0	0	0	0	0
	3	0	0.7	0	0	3.4	0
	4	0	0	0	0	1.0	0

address of a data element by serial operations of SLA (Shift-Left Arithmetic), +, -, and -. It gets input data A, B, C, and D from four PUs 0, 1, 2, and 3, and produces the output $D - \{C - (2A + B)\}$ at PU3. The results prove the effectiveness of the operation in the areas of emulation and language processing. The results also taught us that the more carefully the user inspects his microprogram, the more chances he has of replacing a combination of transfers and numerical/nonnumeric operations with a single serial operation.

8.2.4 Partial Join Mode

As described in subsection 3.4.1, MUNAP has two modes for the activation of nanoprograms. The all join mode has a simple control scheme: a microinstruction always waits for the end of all the nanoprograms activated by the preceding microinstruction. The partial join mode has a looser condition than the all join mode in that the microinstruction can activate nanoprograms before the end of all the nanoprograms being executed.

Our experimental results on the two modes are summed up as follows:

Necessity of the partial join mode: The Prolog project showed that the implementation of the unification parallelism needs the partial join mode so that micro- and multiple nanoprograms can run independently. The loose coupling enables each micro- or nanoprogram to execute its own program, being independent of the other programs, as well as to interact with them when necessary. This was evidenced by the use of the MNFC microinstruction, as

will be described later.

Effect of the partial join mode: We evaluated the effect of the partial join mode by running the same microprogram in the two modes. In the case of the Smalltalk programs, the effect was a 1.3 to 1.5% reduction in the execution microsteps. The reason for the low reduction is that the depth of nanoprograms is very shallow (1.13 - 1.16 from the experiment). Here, the depth means the longest machine cycles among multiple nanoprograms activated by a microinstruction. We can expect better performance improvement if there is greater depth because the waiting time of the succeeding microinstruction is proportional to the depth.

8.3 Nonnumeric Functions

When we designed MUNAP, we defined the microfunctions with a view toward nonnumeric oriented applications. The selected functions are as follows: data permutation operations by the shuffle exchange network (SEN), logical addresses supported by the combination of the SEN and address modifier (AM), data division and concatenation operations by the divide and concatenate unit (DCU), and bit operations by the bit operation unit (BOU). Table 8.3 shows our experimental results. Notice that the numerical computation is divided into two columns for FFT and LU decomposition because the two microprograms showed quite different characteristics for several items.

Table 8.3
Nonnumeric unit utilization^a

	A	B	C	D	E	F	G
Total execution steps($\times 10^3$)	323	1186	1340	1099	857	1182	632
Shuffle exchange network							
Total utilization	84.7	69.3	80.4	76.3	70.0	97.6	93.7
Shift	40.6	33.8	62.2	75.5	65.6	19.6	55.3
Exchange	0.0	0.0	1.9	0.0	2.8	17.7	0.0
Broadcast	22.8	8.3	27.4	6.9	16.1	0.0	41.5
Functional control	36.5	57.9	8.5	17.6	15.5	62.7	3.2
MM interleave access							
Total utilization	5.7	10.0	2.0	13.2	5.0	5.7	14.2
1 byte	0.0	10.6	68.8	0.0	44.5	0.0	0.0
2 bytes	99.7	0.0	0.2	100	55.5	100	0.0
4 bytes	0.0	89.2	1.1	0.0	0.0	0.0	100
8 bytes	0.1	0.2	29.6	0.0	0.0	0.0	0.0
ALU total utilization	23.4	44.5	34.9	48.0	39.1	47.0	51.0
Data divide and concatenate							
Total utilization	22.0	1.8	8.0	9.5	11.9	12.2	7.0
Concatenation	0.3	18.5	18.5	11.5	29.1	29.2	33.2
Division	99.7	81.5	81.5	88.5	70.9	70.8	66.8
Functional Control	0.0	0.0	1.1	0.0	0.0	20.6	29.9
Bit processing							
Total utilization	5.4	0.6	0.9	2.0	4.0	6.5	7.5
Set/Reset	3.8	19.0	41.9	46.1	41.2	44.4	50.3
Test	96.2	79.9	57.4	53.9	58.2	28.6	26.0
Priority encoding	0.0	1.0	0.7	0.0	0.6	27.0	23.7
Functional control	0.1	2.3	0.0	0.0	0.0	0.0	8.5

a. key: A, Emulator; B, L^b; C, MSDL; D, Prolog; E, Smalltalk; F, FFT;
G, LU decomposition.

8.3.1 Shuffle Exchange Network Functions

The total utilization of the SEN was very high. The shift, broadcast, and functionally controlled operations occupy most of the usages. The shift was used for data exchange between the PUs, alignment of data transferred between the MM and PUs (see figures 3.4 and 3.5), and serial PU operations. The broadcast was used to distribute a variable or other control information in the PU to the other PUs so that the PUs can work in parallel. For example, in the ECLIPSE emulator, the operation code was decoded in PUO and the result was broadcast to the other PUs by the function. Most of the functionally controlled operations were specified by the address modifier so that the data read from the MM can be aligned. Special functions were utilized in unexpected areas. An irregular exchange of four 16-bit data ABCD to DCBA was used to implement two stacks on an SPM area. Sixteen-bit mirror transform was used for bit reversal of FFT.

Consequently, the shuffle-exchange network functions were well utilized for various applications.

8.3.2 Address Modifier Functions

The total utilization changes depended on the applications. L⁶ and FFT frequently accessed data on the MM. Mainly the normal mode was used, but the unit of the access has a partiality. The definition of the data structures basically determines the length. For example, the ECLIPSE emulator (A) used 2 bytes because the word length of ECLIPSE is 16 bits. MSDL (C) used 1- and 8-byte access for data and the symbol table, respectively.

Prolog (D), FFT (F), and LU decomposition (G) use 2, 2, and 4 bytes, respectively, based on their units of data words. The other applications do not show such one-sidedness. For example, in the Smalltalk-80 (E), 1 and 2 bytes are used evenly because the units of instruction and data were defined as 1 and 2 bytes, respectively. Thus, variable length word accessing proved to be useful not only for applying the machine to a wide variety of applications but also for realizing various data structures within a single application.

The skewed and direct modes were not effectively utilized. The skewed mode was designed for two-dimensional accesses. However, the fixed length of the access unit (1 byte) seemed to make the application quite difficult.

8.3.3 Division and Concatenation Functions

The total utilization of the DCU unit shows a good usage ratio, compared to a general purpose ALU. The functions were especially useful for decoding a machine instruction in the emulator, decomposition of tagged data and other data manipulations in language processors (such as MSDL, Prolog, and Smalltalk), and decomposition of numerical data in FFT and LU decomposition. In each application, division is far better utilized than concatenation. However, most of the operations are simple, and the complex operations, like DCU stack operation, were not as heavily utilized as we expected. The functional control of the DCU was mostly utilized not for function specification but for bit position and shift amount specification. That is, using the

scheme, computed results were set to the functional control registers to determine the position to be divided or concatenated at run time. The effect of the DCU was evaluated by describing the DCU operations by the ALU. This proved 40% and 30% reduction in total execution steps in the emulator and MSDL, respectively.

8.3.4 Bit Operation Functions

The total utilization was not so high as for the DCU. Mainly the set, reset, and test functions were used. The priority encode was used for special applications. The bit count was not used. The bit set, reset, and test functions were used for handling the tag field and sign bit. The priority encoding was used to normalize floating point numbers in numerical computations. We would say that the utilization of the bit count function required effort because most users were not accustomed to this kind of function. The BOU functions, such as set, reset, and test, may be replaced by 1 to 2 ALU operations; the priority encode was replaced by 10 ALU operations. Thus, the effect of the BOU was not as great as that of the DCU. The maximum effect was for the FFT microprogram (15%).

8.4 Two-Level Microprogramming

8.4.1 Number of Nanosteps

The average number of nanosteps activated by one microinstruction indicates a flexibility of two-level microprogramming because the one-level control may be viewed as a special case of two-level

micropogramming where the micro has a one-to-one relationship with multinanoinstructions in multiprocessors and the nanostep is always 1. The experimental results show that the nanosteps for MIMD type processing were around 1.1 to 1.5, while those for SIMD type processing were 7.7 (FFT) to 78.5 (LU decomposition). The reason is that the data in four PUs for MIMD processing are mutually dependent and the result of other PUs is needed after a few steps of nanoprogam execution. In SIMD type processing, the PUs are mutually independent. Thus, once necessary data are fetched from the MM to the SPM, the nanoprogams can continue their execution for a relatively long time period. The control is returned to microlevel only when accessing the MM or controlling sequences, such as the call/return micropogram subroutine.

8.4.2 Nanoprogam Compaction

The effect of nanoprogam compaction has been described in section 6.4. Our application of the optimizing loader to the MSDL interpreter confirmed those results. The 140 modules of two-level micropograms were linked to a large load module of 3.4 K micropogram and 2.1 K x 4 nanoprogams. After two applications, we obtained a maximum assignment ratio (AMR) (see subsection 6.4.2 for the definition) of 86%. Thus, the two-level micropogramming scheme allows high utilization of nanoprogam memory, while this kind of optimization is impossible for a one-level micropogramming scheme.

8.4.3 Usage Frequencies of Micro- and Nanoinstructions

Tables 8.4 and 8.5 show the dynamic usage frequencies of micro- and nanoinstructions. We shall summarize the features on the basis of these results.

1. Microlevel characteristics: The characteristics are summarized as follows: (a) the sequence control microinstructions (BN, B, TBN) represent 50-80% of the

Table 8.4
Usage frequencies of dynamically executed microinstructions^a (unit: %)

Microinstruction (instruction name)	Emulator	L ⁶	MSDL	Prolog	Small-talk	Numerical Computation
Data transfer (AA,AB,BA) ^b	19.9	19.1	25.3	33.6	36.3	15.4
Constant generation (LT)	0.0	0.0	0.5	1.6	0.9	0.0
MM address generation (AM) ^b	5.7	10.0	2.0	13.2	5.3	5.0
Sequential & unconditional branch (BN) ^b	20.9	21.0	23.1	13.2	14.9	36.8
Subroutine call/return, & indirect branches (B)	38.1	19.2	29.4	26.2	27.5	38.6
Conditional branch (TBN) ^b	15.3	30.7	19.6	10.5	15.1	4.4
Micro-nano flag control (MNFC)	0.0	0.0	0.0	1.7	0.0	0.0

a. CB (a data transfer between microlevel facilities), and TB (unconditional branch that does not activate a nanoinstruction) were not used.

b. The microinstructions activate nanoinstructions, but the others do not.

instructions, (b) data transfer microinstructions (AA, AB, BA) have the second highest frequency of 20-36%, and (c) MM access is around 5%, except 10% for L⁶ and 13% for Prolog. The first item indicates the importance of microlevel sequencing functions in the two-level microporgramming scheme. The second data transfer operations were required to support parallel operations in four PUs. The small percentages for MM accesses prove the effectiveness of a large capacity scratchpad memory in each PU. In the cases of L⁶ and Prolog, the relatively large percentages were due to the fact that the MM contains a large capacity of blocks for L⁶ and data stacks for Prolog.

The LT, CB, TB, and MNFC show small percentages. The small usage ratios of the LT instruction indicate that the literal generation was done by nanolevel literal instruction, NLT, because it allows the user to specify four independent 16-bit literals in four PUs, and literals are usually used inside the PUs. However, the use of the LT occurred when the generated literals were used at microlevel. The example concerns setting the literals to the microstacks (MSTK). The CB and TB microinstructions were scarcely used. The reason is that the functions may be covered by the other microinstructions. This issue will be discussed later as the orthogonality of the instructions. The MNFC was only used in Prolog for implementing the microarchitecture level distributed processing. The instruction allows the microprogram to act as the coordinator of multiple nanoprogams. Thus, although the percentage was very small, MNFC instruction played an important role in realizing an

Table 8.5
Usage frequencies of dynamically executed nanoinstructions (unit: %)

Nanoinstruction (instruction name)	Emulator	L ^s	MSDL	Prolog	Small- talk	Numerical computation
ALU control (ALU)	23.4	44.5	34.9	48.0	38.9	49.0
DCU control (DCU)	22.0	1.8	8.9	9.5	10.5	9.6
BOU control (BOU)	5.4	0.6	0.9	3.0	4.0	7.0
External control (EX)	25.0	21.4	23.1	16.7	30.9	2.9
Constant generation (NLT)	8.9	16.0	17.9	9.5	6.6	7.0
Conditional branch (NTB)	13.1	14.9	13.8	12.8	9.0	21.7
No-operation (NOP)	2.1	0.7	0.5	0.4	0.0	2.9

interesting concept.

2. Nanolevel characteristics: The characteristics are summarized as follows: (a) ALU nanoinstructions have the highest frequency, 23 to 50%, (b) the second highest frequencies, around 20% went to external controls (EX), (c) the nanolevel sequence controls (NTB) were around 13%, (d) constant generations (NLT) occupy 7-18%, and (e) the frequencies for DCU and BOU operations vary according to applications. The reason for high utilizations of ALU operations and external operations is trivial. They were used for usual processing and for data transfers between processor units, respectively. It is surprising that the nanolevel sequence control function was very useful. The reason is twofold: (i) the NTB nanoinstruction was combined with the TBN microinstruction to detect the PU status, transfer the result to microlevel, and reflect to microlevel sequence control, and (ii) the NTB nanoinstruction plays an important role when long nanoprocessors

run independently. An example of the latter case is the numerical computation of SIMD type. Nanolevel constant generation was used instead of microlevel constant generation (LT). These results suggest the importance of job sharing between micro- and nanolevels.

8.5 Systematic Approach to Architectural Improvement

8.5.1 Evaluation Method for an Instruction Set

In order to guide architectural improvement systematically, we first looked for general standards for evaluating architectures [YAMA86b]. These were defined as orthogonality, uniformity, and extensibility [MYER78, WULF81]. *Orthogonality* means independence between the instruction functions. Its objectives are (1) to hold the number of basic concepts to a reasonable minimum, (2) to maximize independence among the instruction functions, and (3) to avoid superfluities [MYER78]. If an instruction can be replaced by some other instructions, the height of orthogonality is proportional to the number of instructions. *Uniformity* implies a complete and consistent structure without special or exceptional cases. It is considered far better to provide uniform primitive functions than to provide the solutions themselves [WULF81]. *Extensibility* means the ability to extend an architecture in a compatible fashion. We shall consider the possibility of extention of the micro- and nanoinstruction sets.

We defined the following algorithm for improvement on the basis of these concepts.

- a. Repeat the following steps, (b)-(e), until a new instruction set architecture reaches a satisfactory condition.
- b. For both the micro- and nanoinstruction sets, compute the number of microcycles necessary for replacing each one with other instructions. Let $f(x)$ denote the number for instruction x .
- c. Executing test programs, measure the dynamic frequency of each instruction as well as the frequency for transition patterns of consecutive micro- and nanoinstructions. Let $g(x)$ denote the frequency for instruction x .
- d. Compute $F(x) = f(x) \$ g(x)$, where $\$$ may be replaced by an appropriate operation, such as (but not limited to) multiplication. If $F(x)$ is lower than a predefined threshold value, x becomes a candidate for improvement.
- e. Investigating the instruction set revised by (d), add or delete instructions in the light of uniformity.

8.5.2 Application to MUNAP

We applied the above method, provided $\$$ is multiplication, the threshold value is about 30 (based on our experience), and the transition patterns include two instructions.

The evaluation results are shown in table 8.6. The instructions that do not appear in the table cannot be replaced by the other instructions. They are conceived as having infinite $f(x)$ values. Microinstructions CB, TB, LT and nanoinstruction

Table 8.6

Orthogonality and utilization frequencies between instructions

Instruction (Operation)		Instruc- tion to be replaced	Number of micro- cycles [f(x)]	Dynamic usage frequencies [g(x)] (%)					F(x)
				Emulator	L ⁶	MSDL	Numerical computation		
M	AA (Transfer)	AB,BA	2-6	18.0	18.6	24.7	11.4		72.7
i	(Shift)	ALU	1-64	18.0	18.6	24.7	11.4		590.7
c	CB (Transfer)	AB,BA	2	—	—	—	—		0.0
r	LT (Literal)	AA,NLT	1	0.0	0.0	0.6	—		0.2
o	BN (Branch)	B	2	20.9	21.0	23.1	36.8		50.9
	TB (Test branch)	TBN,B	2	—	—	—	—		0.0
	TBN (Test branch)	TB,NTB	2	15.3	30.7	19.6	4.4		35.0
N	DCU (Divide)	NLT,ALU	2-13	22.0	1.8	8.9	9.6		79.3
	(Concatenate)	NLT,ALU	3-14	22.0	1.8	8.9	9.6		89.9
	BOU (Bit test)	NLT,ALU	1-2	5.4	0.6	0.9	7.0		5.2
	(Bit set)	NLT,ALU	1-2	5.4	0.6	0.9	7.0		5.2
	EX (Transfer)	ALU	1-2	25.0	21.4	23.1	2.9		27.2

BOU were chosen as candidates for the improvement. (For the micro- and nanoinstruction formats, see figures 3.6 and 3.9.) The following are the result of our consideration:

1. **Deletion:** CB and TB should be deleted. The source and destination of the transfer microinstruction CB may be covered by other microinstructions. And CB cannot specify the SEN function explicitly. TB was not used because it cannot activate nanoprocessors to test the flags inside processor units. LT is the next candidate for deletion. The reason for the low frequency is

that nanolevel NLT covers the function, generating 16-bit literals in each PU. Another reason for nano NLT usage instead of micro LT is that literals are usually used for numerical or nonnumeric operations in PUs and it is desirable to have the generator in PUs. However, LT was used for setting a literal to micro stacks, as described earlier.

2. Extension: The definitions of the BOU nanoinstruction could be changed to enhance its usability. The present BOU can set, reset, and test only 1-bit flags. Thus, it may be extended for multiple bits operations. Moreover, the frequent use of floating point operations required the sign extend function for floating point numbers in order to extend their mantissa.

3. Addition: From the viewpoint of uniformity, a function of dynamic loading for the nanoprogram should be provided that corresponds to the MPMW for dynamic microprogram loading.

The notable transition patterns shown in table 8.7 represent 70% of all possible transition patterns. The results suggest the possibility of the following improvements:

1. Synthesis of BN - TBN microinstruction pair: The high usage ratio of this pattern indicates the importance of the following operations. The BN activates nanoprogams to do some operations by ALU, DCU, or BOU. The succeeding TBN activates NTB nanoinstructions to have them test the result of the operations and set the micro-nano flags (MNFL), and makes a microlevel

Table 8.7

Transition patterns of micro- and nanoinstructions (unit: %)

	Transition pattern	Emulator	L^6	MSDL	Numerical computation
Micro	AA-AA	6.3			
	AA-B	8.3	11.7	11.6	
	AM-AA		10.0		5.0
	B-AA			8.2	
	B-B	16.0		6.4	8.3
	B-B N	15.3	14.7	12.6	29.5
	B N-B				25.7
	B N-T B N	11.0	14.2	11.0	6.3
	T B N-AM		13.6		
	T B N-B	7.4		6.5	
Nano	T B N-B N			6.8	
	T B N-T B N		12.3		
	Total	64.3	76.5	63.1	74.8
	ALU-ALU	12.2	45.9	18.5	19.0
	ALU-DCU				22.6
	ALU-NLT		3.1	6.1	
	ALU-NTB				5.6
	DCU-ALU	21.7			
	DCU-DCU	18.5			
	EX-ALU		15.4	10.8	
	EX-NLT			13.8	
	NLT-ALU		22.7	22.1	
	NTB-ALU				20.2
	NOP-NOP	20.3			
Total		72.7	87.1	71.3	67.4

branch according to the MNFL value.

If we could redefine the nanolevel operations as their operation results are always reflected to MNFL in the same machine cycle, the BN - TBN pair would be replaced by one TBN. The new TBN will activate nanoprogams to do some operations, and at the final cycle of the operations it will make a microlevel branch. This redefinition saves one microinstruction as well as one machine cycle.

2. Synthesis of TBN - B microinstruction pair: This case happens when we make an indirect (and subroutine call) branch by the B microinstruction according to the test result of micro-nano flags by the TBN microinstruction.

However, the synthesis of TBN - B microinstructions is quite difficult as they have few common fields and the total length of the synthesized microinstruction will be twice as long as the original one. Instead of this synthesis, we shall be able to synthesize a TB - B pair to a new microinstruction which tests the micro-nano flags and makes a microlevel branch. The difference between TBN - B and TB - B is that the latter cannot activate nanoprogams.

3. Synthesis of NLT - ALU nanoinstruction pair: The high frequencies of NLT-ALU combinations indicates the need for a 16-bit literal generation just before ALU operations.

The NLT - ALU synthesis is impossible if we keep the length of nanoinstructions. And our other experimental results showed 60-90% of literals generated by NLT were within 0 to 7F in

hexadecimals, which can be specified by the 7-bit S field of an ALU nanoinstruction. Thus, we would say that most of the NLT - ALU pair will be replaced by the present single ALU if the microprogrammer tries to generate a constant not by NLT but by the S field of the ALU when the range falls within 0 to 7F.

4. Synthesis of ALU - ALU nanoinstruction pair: The ALU - ALU operations means consecutive ALU executions in a processor unit (PU). They may be dependent or independent.

To realize this synthesis, the PU should have two ALUs. In addition to this, we should solve the problem of how to squeeze necessary information for controlling the two consecutive ALU operations. More precisely, two ALU, S (source), and D (destination) fields are required (see figure 3.9). If the two ALU operations are independent, these fields cannot be reduced. However, if dependent, we can save one S and D by directly connecting the output of one ALU operation to the input of the other ALU operation. This will save not only a space for an ALU nanoinstruction but also time by making unnecessary some store-load operation pairs.

Consequently, our systematic investigation suggests the deletion of two microinstructions (CB and TB) as well as the synthesis of three pairs of micro- and nanoinstructions (BN-TBN, TB-B, and ALU-ALU). The contradictory requirement for the TB microinstruction will be met by absorbing the present TB function into a synthesized microinstruction from the TB-B pair.

9

Future Directions

We have been working on MUNAP for over five years. Related projects have covered most of the crucial issues in the areas of microprogramming and computer architecture. In the application projects, we devoted a lot of effort to the design and implementation of an innovative system with some unique features, utilizing the architectural features of MUNAP. These yielded experimental results that can be a most reliable predictor. Here we shall try to address MUNAP in the light of global computer architecture and clarify how MUNAP architecture can be utilized in this rapidly advancing field.

9.1 Parallelism of MUNAP

9.1.1 MIMD Register Transfer Level Parallelism

The MIMD is most comprehensive, covering both the SISD and SIMD

types. With MIMD parallelism, a parallel machine can easily exploit the parallelism inherent in a given environment. In our applications, the emulations of the machine language and other languages needed about 70% of MIMD type operations. On the other hand, 90% of numerical computation is of the SIMD type. The ratios mainly depend on the uniformity of the data. Thus, a machine to be used in various environments should have the MIMD parallelism that covers both parallelisms. This is the reason why most of the microprogrammable parallel machines, mentioned in section 4.2, as well as our machine fall into the category of MIMD register transfer level parallelism.

9.1.2 Explicit and Implicit Parallelism

Throughout the various applications, we kept in mind the important difference between areas with implicit parallelism and those with explicit parallelism. Emulation and programming language processing belong to the former, while three-dimensional graphics and numerical computation belong to the latter. For example, sequential programming languages, such as Smalltalk-80 and Prolog, do not have explicit parallelism. The inherent parallelism may be exploited by a careful observation of language semantics and processing. Register transfer level parallelism allows the user to make microscopic processes parallel, preserving the semantics of the language. On the other hand, graphics and numerical computations, such as FFT and LU decomposition, have explicit parallelism in that the same operation may be applied to a large amount of uniform data in

parallel. Usually, a problem with explicit parallelism treats uniform data in an SIMD fashion. The implicit parallelism may be further applied to each data element. For example, a data set for each pixel in a graphics application was processed in parallel at the register transfer level.

Explicit parallelism: It is relatively easy to exploit explicit parallelism. Most of the conventional parallel and pipeline processors utilize this parallelism. In our case, a typical example is a problem involving numerical computation, such as FFT. The average numbers of dynamic utilization were 3.6 - 3.8 out of 4 PUs. As each PU operates on a data item of large uniform data, the numbers would have been higher had we had more PUs. Thus, in this area, it will be better to increase the number of PUs, utilizing the basic architecture of two-level microprogrammed multiprocessor architecture, or to use a network to connect a number of two-level microprogrammed multiprocessor computer. Notice, however, that the application is limited to areas with explicit parallelism.

Implicit parallelism: To our surprise, we obtained average numbers of 2.1 - 2.5 in applications with implicit parallelism. Thus, we can exploit at least 2 - 3 PU parallelism from an essentially sequential algorithm. And we can conclude that it is appropriate to exploit 4 PU parallelism in an implicit parallelism. We investigated the reasons for the relatively high utilization and found, first, that we can find mutually independent operations from those generated for a small,

seemingly sequential process. This kind of microparallelism may be exploited by microprogrammers when they define an algorithm and describe the microprogram. To help this process, automatic detection techniques of the parallelism are strongly desired. For language processing, the level of intermediate codes should be high enough to allow the extraction of implicit parallelism. Our Prolog and Smalltalk-80 projects proved the effect experimentally. Second, the hardware organization of MUNAP, which allows the user to apply multiple PU operations serially on a single datum in one microcycle. Our experiments show that (1) this type of serial operation is utilized in MIMD type processing, where data transfers between processors are frequently required, and (2) the ratio of a two-stage operation is the highest. Furthermore, had this operation not existed, 12% more steps would have been necessary for an application program. This indicates the importance of the high speed execution of a series of mutually dependent operations. If a parallel computer has this capability, it can exploit maximum parallelism by performing mutually independent operations in parallel and mutually dependent operations in series at a high speed with the help of the network.

9.1.3 Architectural Features for Enhanced Parallelism

Except for the control structure, we believe that the key to realizing a comprehensive architecture that allows the above mentioned exploitation of parallelism is to have (1) a large capacity local memory, (2) an interleaved main memory, and (3) an

interconnection network. The large capacity local memory allows the user to save the necessary part of the data in the CPU and thus avoid frequent accessing of the main memory. Our machine contains four 1 K local memories. The effect is clear in the low usage ratios, 2.0 - 5.7%, of MM access microinstructions in our applications. The second, interleaved, memory realizes parallel access to the MM and provides a variety of word lengths especially beneficial for emulation. The third, the interconnection network, plays the most important role in parallel processing; it can be used to connect, among other things, functional units, fast temporary storages, and large capacity main memory. Our experiment shows 70 - 98% of the dynamic usage ratios of the network functions for the total number of execution steps. There are many possible network configurations. However, we think our choice of a shuffle network with exchange and broadcast cells is successful because of its simple structure and comprehensive functions.

9.2 Application Areas and Microfunctions

9.2.1 Support for Multiple Environment - RISC or CISC

There are two major approaches to having a single machine flexibly support different environments. One is to collect the requirements from the environment itself, sort them, and implement the functions directly by hardware. The machine architecture will look like a collection of tailored instructions for the multiple environment. The second approach is to provide

primitives from which the functions can be synthesized. This issue has common grounds with the RISC-CISC controversy [PATT82], where CISC (Complex Instruction Set Computer) provides an application oriented complex instruction set and RISC (Reduced Instruction Set Computer) provides a primitive reduced instruction set. The difference is that our discussion is not at the machine language level but at the microarchitecture one.

Our basic principle of "Provide primitives" was similar to RISC. Enriching nonnumeric functions, we tried to make the machine comprehensive. Special attention was paid to keeping the composability of the functions; i.e., a required function may be composed from the primitive operations at a reasonable cost. The experimental results proved that uniform primitive functions for numerical and nonnumeric processing were utilized well, as described in chapter 8. We could say that our "provide primitive" principle proved to be very attractive and provides an experimental foundation for the RISC concept.

9.2.2 Principles for Primitives Definition

This subsection describes the major issues involved in defining primitives. It is natural that the discussion should cover the same ground as the discussions of design decisions (section 2.3) and architectural evaluations (section 8.5).

Uniformity: Throughout the experiments we found several cases where the loss of uniformity hurts the usefulness of unique operations. For example, the unit of row wise or column wise

access for two-dimensional data is limited to 1 byte. The designer's intention was to utilize a 1 byte access function for accessing multiple bytes of data. However, this was not so easy. In many cases, they utilized some other functions to compose an access function for two-dimensional data of its own size. The second example is the mirror transformation of the network. The limited number of transformation units restricts the applicable situation. The third example is the uniformity between micro- and nanolevels. At present, the microprogram memory is writable, but the nanoprogram one is not. This limits the scope of architectural change. All this suggests that the principle of uniformity is very important at the design stage of a universal host.

Extensibility: As a universal host, our machine was given potential for future extension at both the hardware and firmware levels. This is particularly important when a processor is designed as a research tool. Examples include the main memory and control memory extension, special micro- and nanoinstructions, and unused micro- and nanooperation codes to be used for extension. The PROMs for the network control also have enough room for new permutation patterns; they have been effectively utilized for additional functions.

Orthogonality: We found that several operations were not orthogonal, especially between the microlevel and the nanolevel. For example, literal generation is possible in both levels. The violations of orthogonality tend to result in the distorted use

of either one of the alternative functions.

Not too much semantics: As described above, our experiments confirmed the principle of "Provide primitives, not solutions" [WULF81] at the microarchitecture level. A microoperation with too much semantic content can be used only in limited contexts. For example, we use Am2903 chips for ordinary ALU operation, and they have a set of special functions. As they are tailored to a specific operation, the user cannot utilize them in our applications. Furthermore, we would like to point out the existence of a gap between the designer's intention and the user's understanding. It is very important for the designer of an architecture to try to let users know his intention concerning the use of special functions by showing example programs or microprograms.

9.3 Control Structures

9.3.1 Control Structures for MIMD Register Transfer Parallelism

Flexibility of the control structure is the key to the success of a microprogrammable parallel computer tailored to various application areas. It should correspond to the MIMD register transfer level parallelism, which, as we described before, is desirable for a universal parallel architecture. The control structure for this type of parallelism usually employs relatively long, horizontally encoded microinstruction formats, where each field controls its corresponding resource. The more the resources activated in parallel, the longer the microinstruction. We can

see several examples of this category (see model B in subsection 4.2.1). The control structure of our machine is unique in that it consists of a set of vertically structured microinstructions with multiple sets of vertically structured nanoinstructions in multiple processor units. As each processor unit has a nanosequencer, the nanoprogram can constitute its own sequence, being independent of micro- and the other nanosequences, and thus realizes a very flexible control structure. We shall describe the features of the architecture, comparing it with a horizontally structured long microinstruction architecture.

The flexibility of the control scheme has been further proved by the Prolog project. It realized a microarchitecture level distributed processing (we call this *distributed microprocessing*) to unify arguments in parallel. The micro-nano interaction mechanism, combined with flag control and data transfer micro- and nanoinstructions, has been shown to be effective in implementing the concept.

9.3.2 Flexibility and Distributed Control of a Two-Level Microprogramming Scheme

The key idea of a two-level microprogrammed multiprocessor architecture machine is that the microinstruction can activate multiple nanoprogam streams. They may be mutually dependent or independent. The natural consequence of this scheme is that the hardware units are modular; they have their own controllers and decide their own behavior inside. This is made evident when a microinstruction activates nanoprogams with multiple nanosteps

in many applications, which suggests the importance of distributed control at the microarchitecture level. The major effects of local control (i.e., the nanoprogram can locally process its control) are summarized as follows: (1) the closeness between the control and the controlled parts, (2) the modularity of the control, and (3) the efficiency of control memory space. The first factor causes the length of the control signal lines to shorten and thus allows faster control. The second factor is indicated by the small number of control signals connected between microlevel and nanolevel (i.e., PU). Basically, signal lines for nanoaddress (12 bits), activation (1 bit), and data lines for input (16 bits) go into each PU. Signal lines for nano halt indication (1 bit), nano test result (1 bit), nano request (1 bit), and data lines for output (16 bits) go out from each PU. These are much smaller than the number of control signals generated inside the PU. Furthermore, they may be repetitively generated inside the PU without the specification from the microlevel. This makes the total amount of control words smaller than for a horizontal one, provided that the contents of microprograms are the same. As to the third factor, the scheme essentially requires less space than a horizontal one because it does not use a PU operation code if the PU is not activated, while the code is necessary if we employ a horizontal microinstruction format. This contributes to the shortening of the length of combined micro- and nanoinstruction words. The third factor enables a compaction of the control part. These factors are advantageous for VLSI implementation, where the

transfer speed of data and control signals as well as the compaction of chip area, rather than the computing speed, have profound implications.

9.3.3 Job Sharing between Micro- and Nanoprograms

Our design decisions for the job sharing were the major data flow control and sequence control by micro, the PU data flow and computation by nano, and the sharing of the interaction between the two levels. For the most part, the job sharing succeeded. However, our experimental results for the sequence control function show that (1) the sequence control microinstructions occupy 70 - 80% of the total microsteps and (2) the sequence control nanoinstructions occupy 20 - 25% of the total nanosteps in the applications where the problems have explicit parallelism and the nanoprograms can continue their sequences without interacting with the other micro- and nanoprocesses. The results confirmed the need suggested by the users to enrich the nanosequencer functions so that each nanoprogram can easily control its own nanoprocesses. Thus, new nanoinstructions, such as call/return and functional branch, should be added to the present sequence control nanoinstructions for unconditional and test branches.

9.3.4 Functional Control

In a situation where certain functional units perform similar operations and where their difference may be controlled by the

functional control register, this scheme provided substantial savings of execution steps. Examples are specification of functions for the network, bit positions for the functional units for bit operations, and division and concatenation (see table 8.3). As it is important for a universal computer to be easily tailored to a given environment, our experiments proved that this scheme may be effectively utilized to "set up" a specific environment for repeated use of similar operations. However, the distorted use of specific setup registers suggests that we should select the functions to be functionally controlled very carefully.

9.4 Development and Evaluation of Two-Level Microprograms

9.4.1 Firmware Engineering

Throughout the experimentation we regarded firmware engineering technology as a key to the success of microprogrammable parallel computers by making possible the development of a large number of microprograms that are efficient on both run time and control memory space. Had we failed to develop appropriate tools for firmware development, we could not have been able to utilize the machine. The technology includes the description, translation, verification, and evaluation of microprograms. In our case, a special language was designed to facilitate the two-level, parallel microprogram description. To develop large microprograms, they should be relocatable. We developed a new optimization algorithm for nanoprogram memory compaction, which

attained 80 - 90% utilization of nanoprogram memory. This is equal to the computed maximum of allocation ratios. The experimental results also showed the possibility of optimization, related to the usage of literals. We prepared special micro- and nanoinstructions for literal generation. However, the bit lengths of 60 - 95% of the literals were within 7 bits, which can be covered by the S field of operational nanoinstructions. Thus, most of the literal generation micro- or nanoinstructions may be merged with the succeeding operational instructions. The evaluation of microprograms includes the execution steps and the usage frequency of micro- and nanoinstructions. Throughout the evaluation, the users pointed out the necessity for making the evaluator faster and for counting the usage frequency of the resources. The former will need a hardware support. This leads to a new idea, *architecture tuning*, where the results of evaluation are automatically fed back to tune the universal computer with the environment.

9.4.2 Enhancement of Reliability

We need a good tool for debugging hardware, firmware, and software. The low reliability of a machine makes users lose their will to develop a large number of microprograms. Our firmware debug support is a debugger, which enables the user to use such commands as break/step run and load/save microprograms. The software debug support is a tagged architecture, where erroneous operations are detected by the semantics of data tags. The lack of support for hardware errors was pointed out by the

users. It is serious because the debugging of functions at a given level includes those at lower levels. Thus, an expert system for microdiagnostics is now under development.

9.5 Universal, Special-Purpose, and General-Purpose

The difficulty of constructing a general purpose parallel processor is widely acknowledged. The emergence of a variety of programming languages and the advances in application areas are now demanding high performance computer architectures that are amenable to their own requirements. Every application or programming language processing requires a machine that is tailored to its parallel algorithms. The development of semiconductor technologies also accelerates the design and development of special purpose parallel processors. On the other hand, general purpose computers have been well tailored to the usual environments, such as running FORTRAN programs very fast. Thus, it seems to be difficult for a universal host to find areas in which to perform. However, we believe that such machines can perform in the following situations to advantage: (1) the case where a computer needs to support multiple environments, such as multiple languages, and multiple applications at a reasonable cost, that is, where a general purpose machine cannot attain adequate performance, yet special purpose machines are not economically feasible (a *workstation* is an example of this type), (2) an environment where problems with an essentially sequential feature or a small parallelism are treated, and (3) a research

oriented environment where various exploration work needs to be tested on a machine with flexible architecture.

Furthermore, we would like to suggest that the firmware-hardware combined approach for the implementation of an application oriented, special purpose architecture is a promising direction. The largest drawback of a special purpose computer is its stiff architecture. Even if its application area is limited, it should allow room for the invention of new algorithms and technological changes in a continually advancing field. If a machine is really special, the architects have to catch up with new situations by continually designing new machines. We should remember Wulf's observation [WULF81] that it is still more expensive to design hardware than to design software doing the same job, unless the job is very simple. We cannot ignore the accumulation of software and firmware. Their total development cost usually exceeds that of hardware. This explains why it does not make sense to develop a new special purpose machine for every technological change. Thus, even a special purpose machine should be expected to be able to respond to a new environment. This, we would say, is another form of universal host: universal in a limited scope for a single application. The firmware will be a promising tool for providing flexible yet high performance architecture for this situation.

9.6 MUNAP as a VLSI Architecture

9.6.1 Level of Integration and Possible Configurations

A two-level microprogrammed multiprocessor architecture is very

attractive as a VLSI architecture. The reasons are (1) modularity of the control and controlled part, (2) shortness of control signal lines, and (3) compactness of the control part. The following are promising applications of MUNAP to VLSI architectures:

1. **Assembling heterogeneous functions:** For small scale integration, functional modules both at micro- and nanolevels may be encapsulated in single chips and may be combined so as to be controlled by micro- and nanoprocessors. In this case, a chip may contain not only the functional unit but also the microorder decoder. Such chips will simplify the assembling process of several kinds of chips without losing modularity between the functions. This configuration also allows a heterogeneous function distribution by encapsulating different functions in chips to be assembled, as shown in **figure 9.1A**. The development of VLSI technologies may allow the total architecture, shown in the figure, to be encapsulated in one chip.

2. **Network connected MUNAPs:** If a two-level microprogrammed multiprocessor configuration is implemented on one chip, we can use it as a component of VLSI network architecture, like the systolic array [MEAD80]. **Figure 9.1B** depicts this configuration. This configuration has "two levels of parallelism," that is, intra- and interchip parallelisms. Our applications to programming languages, such as Prolog, suggested the importance of this kind of parallelism combination.

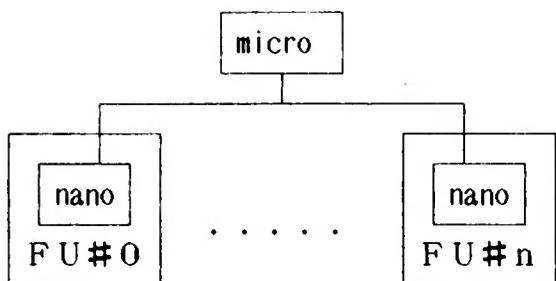


Figure 9.1A
Promising application of MUNAP architecture:
distribution of heterogeneous functional units (FUs)
with nanoprogram memory .

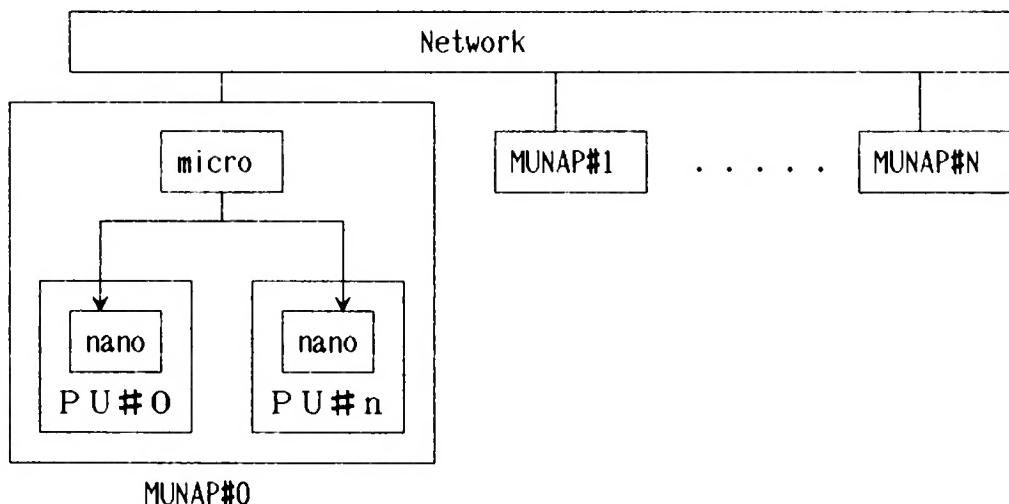


Figure 9.1B
Promising application of MUNAP architecture: multiple MUNAPs.

3. Large number of PUs with nanoprogram memory: As we can control any number of processor units under the two-level microprogramming scheme, a large number of one-, two-, three-dimensionally structured processor units may be controlled as shown in figure 9.1C. The example is the Connection machine [DANI85]. Having a nanoprogram, each processor unit has the capability of flexible, and distributed control.

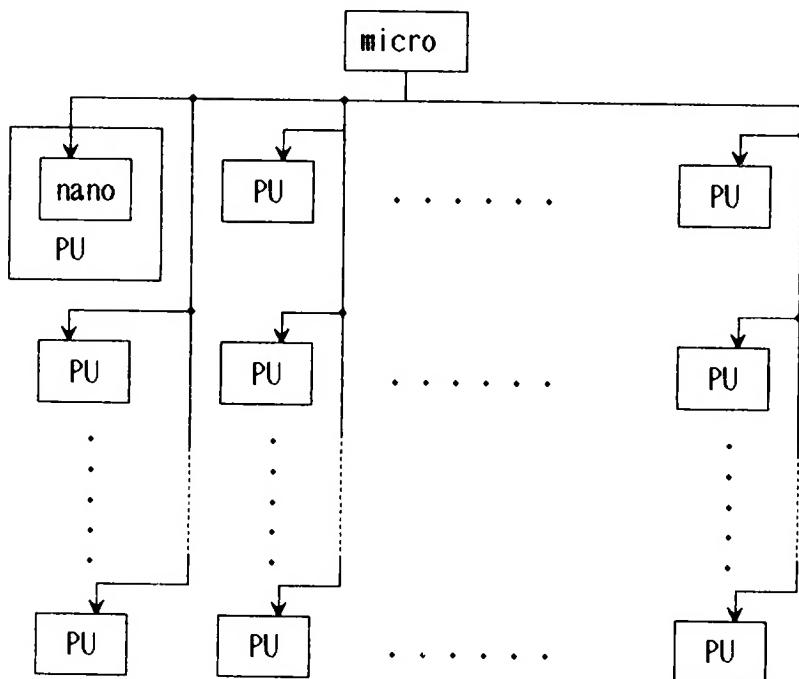


Figure 9.1C
Promising applications of the MUNAP architecture:
large number of PUs with nanoprogram memory.

Microprocessor chips usually also have this capability. For example, Z80 chips may be controlled according to this scheme; in this case, a start program address for each Z80 is to be specified by a higher level controller that corresponds to microlevel.

9.6.2 Relation to the RISC and VLIW Architectures

Recently, the reduced instruction set computer (RISC) and very long instruction word (VLIW) machine have attracted attention as architectures amenable to VLSI requirements. To explore the future possibilities of the MUNAP architecture this last section takes a quick look at the concepts and considers the relationship

between MUNAP and them.

The major design constraints on the RISC architecture are (1) execute one instruction per cycle, (2) make all instructions the same size, (3) access memory only with load and store instructions, and (4) support high level languages [PATT82]. The simple vertical type instruction sets both at micro- and nanolevels indicate that the MUNAP architecture may be viewed as a complex of two types of RISC chips, one corresponding to micro and the other to multi nanos. At both levels one instruction may be executed in one cycle (constraint 1) and the size of instructions are same (constraint 2). The MM read and write operations correspond to load and store instructions (constraint 3). However, MUNAP does not have direct support for high level language processing (constraint 4). It only provides primitive tools, such as large capacity microstacks and scratchpad memories, for the processing.

The defining properties of VLIW architectures are (1) there is one central control unit issuing a single wide instruction per cycle; (2) each wide instruction consists of many independent operations; and (3) each operation requires a small statistically predictable number of cycles to execute [NIC084]. Precisely speaking, the MUNAP does not satisfy these constraints. It does not have a very long horizontal microinstruction. However, the parallel execution of multiple nanoinstructions may be viewed as a kind of VLIW (very long instruction word) architecture. The microprogram is one central controller (property 1) and multiple nanoprocessors control many independent operations (property 2).

The MUNAP architecture is more comprehensive than the VLIW architecture in that necessary cycles for executing each operation may be controlled flexibly in MUNAP (property 3). The most important similarity is that two architectures have a common objective of taking advantage of *fine-grained parallelism*.

There are still controversies on the effectiveness of the proposed architectures [COLW85]. And the purpose of our discussion here is not to claim that MUNAP is RISC and/or VLIW. Instead, we expect that various discussions of RISC and/or VLIW architectures may be applicable to MUNAP because of the architectural similarities.

The first lesson from the discussions is that the RISC and VLIW concepts have somewhat the flavor of microprogramming technologies applied at the machine language level. The RISC's simple instruction set is quite popular at the firmware level. The horizontally microcoded machines are the most familiar examples of the VLIW architecture. It might be a natural coincidence that D. A. Patterson and J. A. Fisher, who coined the terms, have done remarkable work in microprogramming. Similarly, we may propose the MUNAP as a machine language level architecture. This suggests an interesting configuration in which a computer includes two levels of machine languages, one for a central controller (corresponds to micro) and the other for multiple processors (corresponds to multiple nanos), not only at the firmware level but also at the machine language level.

The second point is that the architectures are closely related to their compilers, which directly generate object codes

runnable on the machines. The language-compiler-machine triplet should be coordinated to utilize the architectural features. For example, a global code optimization technique, called trace scheduling [FISH81], has been developed to allow an effective VLIW machine compiler to be built. On the other hand, our translator-interpreter approach is to define an intermediate code so that we can utilize the parallelism of the MUNAP, and then develop the translator and firmware interpreter. Carefully hand-coded microprograms have been developed to utilize a fine-grained parallelism. To compare the performance of the two approaches, we need to develop a compiler that directly generates time- and space-efficient two-level microprograms from a given high level language program. A global optimization technique may be applicable to the generation phase of the compiler.

All of these suggest our promising research problems for the future.



References

- ADVA79 Advanced Micro Devices, Inc.: *Am2900 Family Data Book with Related Support Circuits* (1979).
- AGRA76 Agrawala, A. K., and T. G. Rauscher: *Foundations of Microprogramming*, Academic, New York (1976).
- AISO80 Aiso, H., H. Iizuka, and K. Sakamura, eds.: *Dynamic Architecture* (in Japanese), Kyoritsu Shuppan Co., Ltd., Tokyo, p. 329 (1980).
- ANDR80 Andrews, M.: *Principles of Firmware Engineering in Microprogram Control*, Computer Science Press, Inc. (1980).
- ARN083 Arnold, C. P., M. I. Parr, and M. B. Dewe: "An Efficient Parallel Algorithm for the Solution of Large Sparse Linear Matrix Equations," *IEEE Trans. Comput.*, Vol. C-32, No. 3, pp. 265-273 (1983).
- ASTR76 Astrahan, M. M., M. W. Blasgen, et al.: "System R: Relational Approach to Database Management," *ACM Trans. Database Systems*, Vol. 1, No. 2, pp. 97-137 (1976).
- BABA80 Baba, T., K. Ishikawa, K. Okuda, and H. Kobayashi: "MUNAP - a Two-Level Microprogrammed Multiprocessor Architecture for Nonnumeric Processing," *Proc. IFIP Congress 80*, Oct. 1980, pp. 169-174 (1980).
- BABA81a Baba, T., K. Ishikawa, and K. Okuda: "Architecture of a Two-Level Microprogrammed Computer MUNAP," *Trans. IECE Japan*, Vol. J64-D, NO. 6 (1981).
- BABA81b Baba, T., and H. Hagiwara: "The MPG System: A Machine-Independent Efficient Microprogram Generator," *IEEE Trans. Comput.*, Vol. C-30, No. 6, pp. 373-395 (1981).
- BABA81c Baba, T., K. Ishikawa, and K. Okuda: "Nonnumeric Processing by a Two-Level Microprogrammed Computer MUNAP," *Trans. IECE Japan*, Vol. J64-D, No. 6 (1981).
- BABA82a Baba, T., N. Hashimoto, K. Yamazaki, and K. Okuda: "Microprogramming Support System for a Two-Level Microprogrammed Computer MUNAP," *Trans. IECE Japan*, Vol. J65-D, No. 10 (1982).

- BABA82b Baba, T., K. Ishikawa, and K. Okuda: "A Two-Level Microprogrammed Multiprocessor Computer with Nonnumeric Functions," *IEEE Trans. on Comput.*, Vol. C-31, No. 12, pp. 1142-1156 (1982).
- BABA83a Baba, T., K. Yamazaki, N. Hashimoto, H. Kanai, K. Okuda, and K. Hashimoto: "Hierarchical Micro-Architectures of a Two-Level Microprogrammed Multiprocessor Computer," *Proc. of International Conference on Parallel Processing*, Aug. 1983 (1983).
- BABA83b Baba, T., K. Yamazaki, N. Hashimoto, H. Kanai, K. Okuda, and K. Hashimoto: "Experimentation with a Two-Level Microprogrammed Multiprocessor Computer," *Proc. of 16th Annu. Workshop on Microprogramming*, Oct. 1983 (1983).
- BABA84 Baba, T., M. Ikeda, K. Yamazaki, and K. Okuda: "Compaction of Two-Level Microprograms for a Multiprocessor Computer," *Proc. of 17th Annu. Workshop on Microprogramming*, pp. 95-104 (Oct. 1984).
- BABA86 Baba, T., T. Sano, S. Suzuki, K. Yamazaki, and K. Okuda: "Software Testing System Supported by a Computer Architecture," *Trans. of IECE Japan*, Vol. J69-D, No. 1, pp. 30-41 (1986).
- BARO81 Baroody, A. J., and D. J. DeWitt: "An Object-Oriented Approach to Database System Implementation," *ACM Trans. Database Systems*, Vol. 6, No. 4, pp. 576-601 (1981).
- BARR73 Barr, R. G., J. A. Becker, W. A. Lidinsky, and V. V. Tantillo: "A Research-Oriented Dynamic Microprocessor", *IEEE Trans. Comput.*, Vol. C-22, No. 11, pp. 976-985 (1973).
- BERN81 Bernhard, R.: "More Hardware Means Less Software," *IEEE Spectrum*, Dec. 1981, pp. 30-37 (1981).
- BRIG74 Brigham, E. O.: *The Fast Fourier Transform*, Prentice-Hall (1974).
- CHAR81 Charlesworth, A. E.: "An Approach to Scientific Array Processing: The Architectural Design of the AP120-B/FSP-164 Family," *IEEE Computer*, Vol. 14, No. 9, pp. 18-27 (1981).
- CHU82 Chu, Y., K. Itano, Y. Fukunaga, and M. Abrams: "Interactive Direct-Execution Programming and Testing," *Proc. of COMPSAC*, Chicago (1982).

- CHUR70 Church, C. C.: "Computer Instruction Repertoire - Time for a Change," *AFIPS Conf. Proc. SJCC*, pp. 343-349 (1970).
- COLW85 Colwell, R. P., C. Y. Hitchcock III, E.D. Jensen, H. M. B. Sprunt, and C. P. Kollar: "Computers, Complexity, and Controversy," *IEEE Computer*, Vol. 18, No. 9, pp. 8-19 (1985).
- DANI85 Daniel, H. W.: *The Connection Machine*, The MIT Press Series in Artificial Intelligence, p. 190 (1985).
- DATA77 Data General Co.: "ECLIPSE S/130 Microprogramming WCS Feature" (1977).
- DATE82 Date, C.J.: *An Introduction to Database Systems*, Third Edition, p. 574, Addison-Wesley, Reading, MA (1982), Vol. II, p.383 (1982).
- DOI86 Doi, H.: "Parallel Processing of an Object-Oriented Language on a Two-Level Microprogrammed Computer MUNAP," M.S. Thesis, Dept. Inf. Sci., Utsunomiya Univ. (1986).
- DOI87 Doi, H., K. Yamazaki, S. Ookawa, T. Baba, and K. Okuda: "Fast Processing of an Object-Oriented Language by a Low-Level Parallel Computer MUNAP," *Trans. IECE Japan*, Vol. J70-D, No. 2 (to be published) (1987).
- FENG81 Feng, T. Y.: "A Survey of Interconnection Networks," *IEEE Computer*, Vol. 14, No. 12, pp. 12-27 (1981).
- FISH81 Fisher, J. A.: "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Comput.*, Vol. C-30, No. 7, pp. 478-490 (1981).
- FISH83a Fisher, A. L., et al.: "Architecture of the PSC: A Programmable Systolic Chip," *Proc. 10th Annu. Int. Symp. on Comput. Arch.*, pp. 48-53 (1983).
- FISH83b Fisher, J.A.: "Very Long Instruction Word Architectures and the ELI-512," *Proc. 10th Annu. Int. Symp. on Comput. Arch.*, pp. 140-150 (1983).
- FISH84 Fisher, J. A.: "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer*, Vol. 17, No. 7, pp. 45-53 (1984).
- FOLE82 Foley, J. D., and A. Van Dam: *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA (1982).
- FRIE77 Frieder, G., and J. Miller: "An Analysis of code

- Density for the Two-Level Programmable Control of the Nanodata QM-1," Proc. 10th Annu. Workshop on Microprogramming, pp. 26-32 (1977).
- FUCH85 Fuchi, K., and N. Suzuki: *Programming Language and VLSI*, Iwanami (in Japanese) (1985).
- FULL76 Fuller, S. H., V. R. Lesser, C. G. Bell, and C. H. Kaman: "The Effects of Emerging Technology and Emulation Requirements on Microprogramming," *IEEE Trans. on Computers*, Vol. C-25, No. 10, pp. 1000-1009 (1976).
- FURU82 Furuya, T., and H. Iizuka: "Static Characteristics of Microprograms in a Microprocessor (PULCE)," *Trans. of IECE Japan*, Vol. J65-D, No. 2, pp. 258-265 (1982).
- GERR80 Gerrity, G. W.: "Hardware Detection of Undefined References," *Comput. Arch. News*, Vol. 8, No. 2, pp. 8-11 (1980).
- GOLD83 Goldberg, A., et al.: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA (1983).
- HAGI80 Hagiwara, H., S. Tomita, S. Oyanagi, and K. Shibayama: "A Dynamically Microprogrammable Computer with Low-Level Parallelism," *IEEE Trans. Comput.*, Vol. C-29, pp. 577-595 (1980).
- HANS73 Hansen, P. B.: *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ (1973).
- HILL81 Hill, D. D.: "A Hardware Mechanism for Supporting Range Checks," *Comput. Arch. News*, Vol. 9, No. 4, pp. 15-21 (1981).
- HOSH83 Hoshino, T., T. Shirakawa, T. Kamimura, T. Kageyama, K. Takenouchi, H. Abe, S. Sekiguchi, Y. Oyanagi, and T. Kawai: "Highly Parallel Processor Array "PAX" for Wide Scientific Applications," *1983 Intern. Conf. on Parallel Processing*, pp. 95-105 (1983).
- HUSS70 Husson, S.S.: *Microprogramming: Principles and Practices*, Prentice-Hall, Englewood Cliffs, NJ (1970).
- IDA77 Ida, T., and E. Goto: "Performance of a Parallel Hash Hardware with Key Deletion," Proc. IFIP Congr. 77, pp. 643-647 (1977).
- IIZU76 Iizuka, H., and T. Furuya: "An Architectural Design of a Microprocessor," *Trans. IECE Japan*, Vol. J59-D, pp. 188-195 (1976).
- INAG87 Inagawa, M., T. Baba, K. Ishikawa, K. Yamazaki, and K.

- Okuda: "Unification Parallelism for Prolog Processing," *Trans. IECE Japan*, Vol. J70-D, No. 2 (to be published) (1987).
- JOHN82 Johnson, M.: "Some Requirements for Architectural Support of Software Debugging," *Comput. Arch. News*, Vol. 10, No. 2, pp. 100-106 (1982).
- KATA83 Katayama, T., and K. Kiriyama: *MUNAP Utility System*, BS Thesis, Dept. Inf. Sci., Utsunomiya Univ. (1983).
- KERN78 Kernighan, B. W., and D. M. Richie: *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ (1978).
- KNOW65 Knowlton, K. C.: "A Fast Storage Allocator," *Commun. Ass. Comput. Mach.*, Vol. 8, pp. 623-625 (1965).
- KNOW66 Knowlton, K.C.: "A Programmer's Description of L6," *Commun. Ass. Comput. Mach.*, Vol. 9, pp. 616-625 (1966).
- KNUT73 Knuth, D.E.: *The Art of Computer Programming*, Vol.3: *Sorting and Searching*, Addison-Wesley, Reading, MA (1973).
- KOWA85 Kowalik, J. S., ed.: *Parallel MIMD Computation: The HEP Super Computer and Its Applications*, MIT Press Series in Scientific Computation, p. 411 (1985).
- KRAS83 Krasner, G.: *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA (1983).
- LAND76 Land, T., and H. S. Stone: "A Shuffle-Exchange Network with Simplified Control," *IEEE Trans. Comput.*, Vol. C-25, pp. 55-65 (1976).
- LAWR75 Lawrie, D. H.: "Access and Alignment of data in an Array Processor," *IEEE Trans. Comput.*, Vol. C-24, pp. 1154-1155 (1975).
- LESS71 Lesser, V. R.: "An Introduction to Direct Emulation of Control Structures by a Parallel Micro-Computer," *IEEE Trans. Comput.*, Vol. C-20, pp. 751-764 (1971).
- MCGR76 McGregor, D. R., R. H. Thomson, and W. N. Dawson: "High Performance Hardware for Database Systems," in *Systems for Large Databases*, North-Holland, pp. 103-116 (1976).
- MEAD80 Mead, C. A., and L. Conway: *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, p.396 (1980).
- MOTO79 Moto-oka, T.: "Future Aspects of Computer Systems," *IECE Japan*, Vol. 62, pp. 1204-1207 (1979).
- MYER78 Myers, G. J.: *Advances in Computer Architecture*, Wiley,

- New York (1978).
- MYER79 Myers, G. J.: *The Art of Software Testing*, Wiley, New York (1979).
- NEWM79 Newman, W. M., and R. F. Sproull: *Principles of Interactive Computer Graphics*, McGraw-Hill, New York (1981).
- NICO84 Nicolau, A., and J. A. Fisher: "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Trans. Comput.*, Vol. C-33, No. 11, pp. 968-976 (1984).
- OHTA84 Ohtani, T., T. Baba, M. Inagawa, T. Iwasaki, K. Okuda, and K. Yamazaki: "Data Abstraction Technique for Relational Database Query Processing," *Trans. IECE Japan*, Vol. J-67-D, No. 12, pp. 1450-1457 (1984).
- OPLE67 Opler, A.: "Fourth Generation Software," *Datamation*, Vol. 13, No. 1, pp. 22-24 (1967).
- PATT82 Patterson, D.A. and H. S. Sequinm: "A VLSI Risc," *IEEE Computer*, Vol. 15, No. 9, pp. 8-21 (1982).
- POTT85 Potter, J. L., ed.: *The Massively Parallel Processor*, MIT Press Series of Scientific Computation, p. 304 (1985).
- REIG72 Reigel, E. W., U. Faber, and D. A. Fisher, "The Interpreter - a Microprogrammable Building Block System," in *AFIPS Conf. Proc. SJCC*, 1972, pp. 705-723 (1972).
- RIDE81 Rideout, D. J.: "Considerations for Local Compaction of Nanocode for the Nanodata QM-1," *Proc. 14th Annu. Workshop on Microprogramming*, pp. 205-214 (1981).
- ROSI72 Rosin, R. F., G. Frieder, and R. H. Echhouse: "An Environment for Research in Microprogramming and Emulation," *Commun. Ass. Comput. Mach.*, Vol. 15, No. 8, pp. 748-760 (1972).
- SALI76 Salisbury, A. B.: *Microprogrammable Computer Architectures*, Elsevier North-Holland, New York (1976).
- SASA85 Sasaki, H., and T. Ooi: *Emulation of a Minicomputer Eclipse S/130 by a Two-Level Microprogrammed Computer MUNAP*, B.S. Thesis, Dept. Inf. Sci., Utsunomiya Univ. (1985).
- SHIB85 Shibaoka, H., and R. Takahashi: *Numerical Calculation on MUNAP*, B.S. Thesis, Dept. Inf. Sci., Utsunomiya Univ. (1985).

- SHIB80 Shibayama, K., S. Tomita, H. Hagiwara, K. Yamazaki, and T. Kitamura: "Performance Evaluation and Improvement of a Dynamically Microprogrammable Computer with Low-Level Parallelism," *Proc. IFIP Congr. 80*, pp. 181-186 (1980).
- SHIB82 Shibayama, K., T. Nakata, T. Kitamura, S. Tomita, and H. Hagiwara: "Emulations on Some Language Processors on a Universal Host Computer QA-1," *Trans. of IECE Japan*, Vol. J65-D, No. 11, pp. 1374-1381 (1982).
- STON80 Stonebraker, M., E. Wong, et al.: "The Design and Implementation of INGRES," *ACM Trans. Database Systems*, Vol. 5, No. 2, pp. 225-240 (1980).
- STR178 Stritter, S., and N. Tredennick: "Microprogrammed Implementation of a Single Chip Microprocessor," *Proc. of Annu. Workshop on Microprogramming*, pp. 8-16 (1978).
- SUZU84 Suzuki, N., et al.: "Sword32: A Bytecode Emulating Microprocessor for Object-Oriented Languages," *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT (1984).
- SUZU86 Suzuki, S., T. Baba, K. Yamazaki, and K. Okuda: "Three Dimensional Color Graphics System Based on Low-Level List Processing Language," *Proc. 31st Annual Convention of Inf. Proc. Soc. of Japan*, 2K-5, p.1731 (1985).
- TICK84 Tick, E.: "Sequential Prolog Machine: Image and Host Architectures," *Proc. of Annu. Workshop on Microprogramming*, pp. 204-216 (1984).
- TOMI83 Tomita, S., K. Shibayama, T. Kitamura, T. Tanaka, and H. Hagiwara: "A User-Microprogrammable Local Host Computer with Low-Level Parallelism," *Proc. 10th Annu. Int. Symp. Comput. Arch.*, pp. 151-159 (1983).
- TOMI86 Tomita, S., K. Shibayama, T. Nakata, S. Yuasa, and H. Hagiwara: "A Computer with Low-Level Parallelism QA-2: Its Applications to 3-D Graphics and Prolog/Lisp Machines," *Proc. 13th Annu. Int. Symp. Comput. Arch.*, pp.280-289 (1986).
- UNGE84a Unger, D., et al.: "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Annual Symp. on Computer Architecture*, pp. 188-197 (1984).
- UNGE84b Unger, D.: "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," *ACM Software Eng. Notes/Sigplan Notices Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh (1984).

- WALT71 Walther, J. S.: "A Unified Algorithm for Elementary Functions," *AFIPS Conf. Proc.*, Vol. 38, 1971 SJCC., pp. 379-385 (1971).
- WARR77 Warren, D. H. D.: *Implementing Prolog-Compiling Predicate Logic Program*, Vols. 1 and 2, D.A.I. Research Report, No.39-40, Department of Artificial Intelligence, Univ. of Edinburgh (1977).
- WILN72 Wilner, W. T.: "Design of the Burroughs B1700," *Proc. SJCC*, Vol. 40, pp. 489-497 (1972).
- WIRT78 Wirth, N.: *PL360, a Programming Langauge for the 360 Computers*, Prentice-Hall, Englewood Cliffs, NJ (1978).
- WULF81 Wulf, W. A.: "Compilers and Computer Architecture," *IEEE Computer*, Vol. 14, No. 7, pp. 41-47 (July 1981).
- YAMA84a Yamazaki, K., H. Kanai, T. Baba, K. Okuda, N. Hashimoto, and K. Hashimoto: "Development of a System Description Language for a Two-Level Microprogrammed Computer MUNAP," *Trans. of IECE Japan*, Vol. J67-D, No. 1, pp. 149-156 (1984).
- YAMA84b Yamazaki, K., H. Kanai, H. Doi, Y. Ono, T. Baba, and K. Okuda: "A Tagged Architecture for Testing Programs Written in a System Description Language MSDL," *Trans. of IECE Japan*, Vol. J67-D, No. 10, pp. 1202-1209 (1984).
- YAMA86a Yamazaki, K., H. Kanai, T. Baba, and K. Okuda: "Architectural Evaluation of a Universal Host Computer MUNAP," *Trans. IECE Japan*, Vol. J69-D, No. 1, pp. 21-29 (1986).
- YAMA86b Yamazaki, K., T. Baba, K. Okuda, and H. Kanai: "Architectural Evaluation and Improvement of a Universal Host Computer MUNAP," *Proc. IFIP Congress 86*, pp. 779-784 (1986).

Index

A

AA microinstruction 44
AB microinstruction 44
abstract data type 12
ACOS 600S 105, 115, 152
address modifier 40, 241
AFR X & Y 55
all join mode 62, 238
ALU 35, 52
ALU nanoinstruction 56
AM
 See address modifier
AM microinstruction 44
Am2903 52
AMFR 43
AMP machine 87
AMR
 See maximum assignment ratio
AND parallelism 183
AP-120B 87
architecture tuning 267
argument 181
arithmetic and logic unit 35, 52
array processor 9
assignment statement 116

B

B microinstruction 44
BA microinstruction 44
backtrack 183
backward substitution 226
BFR X & Y 55
bit operation 243
bit operation unit 35, 52, 243
bit processing 31, 80
bit reversal 220
bit test and set 222
BN microinstruction 44

body 181

BOU
 See bit operation unit
BOU nanoinstruction 56
broadcast 38, 241
butterfly 219
byte code 198

C

C 54, 149
C-approach 176
CB microinstruction 44
central processing unit 4
CFR X, & Y 55
Church's problem 80, 82
CISC
 See complex instruction set computer
class 197
clause 181
clipping algorithm 212
CLU 227
clustering 185, 193
CM-1
 See Connection Machine
compiled method 199
completeness 30
complex instruction set computer 88, 260
computer architecture 7
computer graphics 210
Condition 6.1 118
Condition 6.2 119
Condition 6.3 120
Connection Machine 87
consistency check 187
control memory 3
CORDIC 214, 220
counter 54
CPU
 See central processing

- unit
 - CX 54
 - CY 54
- D**
 - D field 56
 - dangling pointer 168
 - data abstraction 175
 - data descriptor 164
 - data independence 179
 - database machine 9
 - DCU
 - See divide and concatenate unit
 - DCU nanoinstruction 56
 - debugger 10, 134
 - description 107
 - DFR X, Y, & Z 55
 - direct mode 40, 41
 - distributed microprocessing 263
 - divide and concatenate unit 35, 53, 242
 - division and concatenation function 242
 - dynamic microprogramming 4
- E**
 - ECLIPSE S/130 79, 104, 115, 135, 146
 - edge 211
 - ELI machine 87
 - emulation 4, 11, 146
 - emulator 146
 - end of nanoprogram 56, 60, 71
 - ENP
 - See end of nanoprogram
 - error detection 163
 - evaluator 11, 136
 - EX nanoinstruction 56
 - execution time 141
 - explicit parallelism 256-257
 - extensibility 248, 261
- F**
 - face 211
 - fact 182
 - Fast Fourier Transform 217-224
 - FC field 56
 - FFT
 - See Fast Fourier Transform
- field handling 32, 80
- fill-in 226
- fine-grained parallelism 274
- firmware 4, 7
- firmware engineering 107, 266-267
- flag register 55, 56, 63
- FLR
 - See flag register
- FLRX 55
- FLRY 55
- forward deletion 226
- forward substitution 226
- functional control 265-266
- functional control register 55
- functor 181
- G**
 - garbage collection 205
 - general-purpose 268
 - Generation Scavenging 195
 - generic routine 177
 - global memory 190
 - global optimization 275
 - goal 181
 - graphics processor 9
- H**
 - hardware 7
 - hardwired computer 2
 - hashing 84
 - head 181
 - high level language machine 9
 - HITAC M-200H 85
 - horizontal microinstruction 23
 - host machine 146
- I**
 - I-approach 176
 - I-code 196, 201
 - IC
 - See integrated circuit
 - implicit parallelism 256-258
 - IMPMAR 37, 64
 - INGRES 176
 - INPMAR 64
 - instance 197
 - instance variable 197
 - integrated circuit 53
 - interactive debugging system 164

interleave access 40
 Interpreter 23, 88
 interpreter 181, 198
 IPR 55
 See also port register

J

jump instruction 202

L

L^c 165, 211
 language-compiler-machine triplet 275
 large scale integrated circuit 6, 96
 LIDS 165
 logical schema 177
 LSI 6, 96, 101
 LT microinstruction 44
 LU decomposition 224-232

M

machine cycle 3, 141
 machine independence 108
 macroarchitecture 8
 macroinstruction 154
 main memory 19, 35, 40, 75
 MAR 40
 MASK 43, 64
 mask register 43
 matrix operation 211, 214
 maximum assignment ratio 133
 MC68000 88-89, 104
 memory partial write register 42
 message 196
 method 197
 MI 22, 60
 See also microinstruction
 micro-nano flag 42-43, 58
 micro-nano interaction 26, 194, 190
 micro-nanostatement 113, 116
 microarchitecture 8
 microassembler 10, 114-118
 microcycle 3
 microinterrupt 59
 microinstruction 3, 22
 microinstruction format 46-47
 microoperation 3
 microorder 3

microprogram debugger 10
 microprogram loader 128
 microprogram memory 3, 35, 37
 microprogrammed computer 2
 microprogramming 4
 microrequest 58, 63
 microsequencing 48-50
 microstack 42
 microstatement 113
 MIMD 6, 17, 73, 92, 111, 136, 234, 244, 256
 See also multiple instruction multiple data stream
 MIMD register transfer level parallelism 262-263
 MINT 59, 64
 mirror transformation 38, 222
 MISD
 See multiple instruction single data stream
 MM 35, 40
 MNFC microinstruction 45, 238, 246
 MNFL
 See micro-nano flag
 Model A 85
 Model B 87
 Model C 87
 Model D 88
 MOS 98
 MPM
 See microprogram memory
 MPMAR 37
 MPMW microinstruction 45
 MPW
 See memory partial write register
 MREQ
 See microrequest
 MSDL
 See MUNAP System Description Language
 MSI 98, 101
 MSTK 50
 See also microstack
 multiple instruction multiple data stream 6
 multiple instruction single data stream 6
 multiple processor units 93
 MUNAP 26, 35
 MUNAP prototype 101
 MUNAP System Description

Language 150

I

nano index register 43
 nanoaddress space compaction 120, 127
 nanohalt 58
 nanoinstruction 22
 nanoinstruction format 57
 nanointerrupt 59
 nanoprogram compaction 244
 nanoprogram memory 35, 37, 52
 nanoprogram utilization 118, 127
 nanorequest 58
 nanosequencer 52
 nanostatement 113
 NHLT 58
 nI 22, 60
 See also nanoinstruction
 NINT 59, 64
 NLT nanoinstruction 56
 nonnumeric function 111, 138, 239
 nonnumeric processing 31
 NOP nanoinstruction 58
 normal mode 40, 42
 normalization 226
 not too much semantics 262
 NPM
 See nanoprogram memory
 NREQ 58, 64
 NTB nanoinstruction 56, 63
 numerical computation 13, 216-232
 NXR
 See nano index register

O

object 196
 object field 201
 object memory 198
 object pointer 195, 199, 204
 OF
 See object field
 Oop
 See object pointer
 OPR 55
 See also port register
 optimizaiton 107
 optimization I 118
 optimization II 120

OR parallelism 183
 orthogonality 16, 30, 248, 261

P

parallel hashing 85
 parallel processing 110
 parallel PUs 142
 parallel test 76-77
 partial join mode 62, 238
 phase 69-70
 physical schema 177
 pixel 210
 PL 360 149
 PLM 181
 PLMI 188
 PMD flag 62
 port register 19, 55
 primitive method 197
 priority encoder 53
 procedure 182
 processor unit 18, 24, 35, 37, 50
 Prolog 12, 181
 provide primitives principle 29, 260
 PU
 See processor unit

Q

Q register 52
 QA-1 87
 QA-2 87
 QM-1 23, 88
 question 182
 quick sequential search 84

R

range check 164, 168, 172
 read only memory 4
 reduced instruction set
 computer 260, 272-275
 register file 52, 147
 register transfer level 17
 register transfer level
 parallelism 18, 255
 relational database system 175
 reliability 267-268
 relocatable microprogram
 loader 10
 return instruction 202
 RF

See register file
RISC
 See reduced instruction set computer
RLU 227
ROM
 See read only memory
RTL
 See register transfer level
rule 182

S

S field 56
S-approach 176
scratchpad memory 54, 147
searching 84
semantic gap 8
SEN
 See shuffle exchange network
send instruction 202
SENFR 43
serial operation 65-69, 73, 236
setup register 28
shuffle exchange network 20, 35, 38, 241
signal processor 9
SIMD 5, 72, 92, 110, 136, 234, 244, 256
 See also single instruction multiple data stream
simulator 134
simultaneous linear equation 224
single instruction multiple data stream 5
skeleton term 182
skewed mode 40, 41
Smalltalk-80 12, 195
SOAR 195
software 7
software engineering 163
software testing 163
sorting 84
sparse matrix 226
special-purpose 268-269
special-send instruction 202
SPM
 See scratchpad memory
SSI 98, 101

stack 42
subclass 197
superclass 197
Sword32 195
system description language 149
System R 176

T

table access 33, 83
tagged architecture 11, 150, 161, 164
target machine 146
TB microinstruction 45
TBN microinstruction 45
term 181
test 58
TEST 58, 63
test branch 117
three-dimensional color graphics system 13, 211
timing analysis 69-70
trace scheduling 87, 275
translation 107
translator-interpreter 275
TS1 field 56
TS2 field 56
TSC field 56
two-level microprogramming 16, 21, 137, 243

U

undefined references 164, 168, 172
unification parallelism 184, 191
uniformity 16, 30, 248, 260-261
universal host 4, 16, 217, 233

V

variable length word access 32, 83
vector processor 9
verification 107
vertex 211
vertical microinstruction 23
very large scale integrated circuit 96
very long instruciton word machine 272-275

virtual microarchitecture 109
VLIW 87
 See very long instruction
 word machine
VLSI 25
 See also very large scale
 integrated circuit

VLSI architecture 269-275
von Neumann architecture 144
von Neumann machine 79, 145

V

workstation 268

The MIT Press, with Peter Denning as consulting editor,
publishes computer science books in the following series:

ACM Doctoral Dissertation Award and Distinguished Dissertation Series

Artificial Intelligence, Patrick Winston and Michael Brady,
editors

Charles Babbage Institute Reprint Series for the History of Computing, Martin Campbell-Kelly, editor

Computer Systems, Herb Schwetman, editor

Foundations of Computing, Michael Garey, editor

History of Computing, I. Bernard Cohen and William Aspray,
editors

Information Systems, Michael Lesk, editor

Logic Programming, Ehud Shapiro, editor

The MIT Electrical Engineering and Computer Science Series

Scientific Computation, Dennis Gannon, editor



MIT LIBRARIES

DUPL



3 9080 02226 2742

Date Due

--	--	--

Lib-26-67

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142

Microprogrammable Parallel Computer
MUNAP and Its Applications
by Takanobu Baba

This book takes up the challenge posed by recent advances in theoretical computer science and artificial intelligence that have created a demand for a radically different type of computer architecture. It demonstrates the possibility of register transfer level parallel computing with microprogrammable flexible architecture that can fulfill a wide variety of user requirements, and provides all the necessary technical information for understanding the process of design, development, and evaluation of this innovative MUNAP computer.

After introducing the basic concepts in the computer architecture and microprogramming area, the book describes how the architect goes about selecting microoperations, considering software/firmware/hardware trade-offs and what schemes might be used for interleaved memory access and interconnection networks. Microprogrammed computer models are defined for the evaluation of computers with similar architectures.

Microprogrammable Parallel Computer presents the results of exhaustive experimentation with this new architecture, showing how it can be exploited in current research on emulation of machine language, tagged architectures, high-level language processing, software testing, database systems, three-dimensional graphics, and numerical computation.

Takanobu Baba is an Associate Professor, Department of Information Science, Utsunomiya University, Japan. *Microprogrammable Parallel Computer* is included in the MIT Press Series in Computer Systems, edited by Herb Schwetman.

BABMH
0-262-02263-X